



**EVOLUTIONARY ARTIFICIAL NEURAL NETWORK WEIGHT TUNING TO
OPTIMIZE DECISION MAKING FOR AN ABSTRACT GAME**

THESIS

Corey M. Miller

AFIT/GCS/ENG/10-06

**DEPARTMENT OF THE AIR FORCE
AIR UNIVERSITY**

AIR FORCE INSTITUTE OF TECHNOLOGY

Wright-Patterson Air Force Base, Ohio

APPROVED FOR PUBLIC RELEASE; DISTRIBUTION UNLIMITED

The views expressed in this thesis are those of the author and do not reflect the official policy or position of the United States Air Force, Department of Defense, or the U.S. Government.

AFIT/GCS/ENG/10-06

**EVOLUTIONARY ARTIFICIAL NEURAL NETWORK WEIGHT TUNING TO
OPTIMIZE DECISION MAKING FOR AN ABSTRACT GAME**

THESIS

Presented to the Faculty

Department of Electrical and Computer Engineering

Graduate School of Engineering and Management

Air Force Institute of Technology

Air University

Air Education and Training Command

In Partial Fulfillment of the Requirements for the

Degree of Master of Science

Corey M. Miller, B.S.C.E.

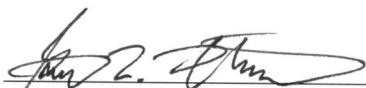
March 2010

APPROVED FOR PUBLIC RELEASE; DISTRIBUTION UNLIMITED

**EVOLUTIONARY ARTIFICIAL NEURAL NETWORK WEIGHT TUNING TO
OPTIMIZE DECISION MAKING FOR AN ABSTRACT GAME**

Corey M. Miller, BS

Approved:



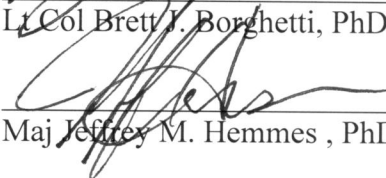
Gilbert L. Peterson, PhD (Chairman)

4 MAR 2010
Date



Lt Col Brett J. Borghetti, PhD, USAF (Member)

4 MAR 2010
Date



Maj Jeffrey M. Hemmes, PhD, USAF (Member)

4 Mar 2010
Date

Abstract

Abstract strategy games present a deterministic perfect information environment with which to test the strategic capabilities of artificial intelligence systems. With no unknowns or random elements, only the competitors' performances impact the results. This thesis takes one such game, Lines of Action, and attempts to develop a competitive heuristic. Due to the complexity of Lines of Action, artificial neural networks are utilized to model the relative values of board states. An application, pLoGANN (Parallel Lines of Action with Genetic Algorithm and Neural Networks), is developed to train the weights of this neural network by implementing a genetic algorithm over a distributed environment. While pLoGANN proved to be designed efficiently, it failed to produce a competitive Lines of Action player, shedding light on the difficulty of developing a neural network to model such a large and complex solution space.

Acknowledgments

I would like to express my sincere appreciation to my faculty advisor, Dr. Gilbert Peterson, for his guidance and support throughout the course of this effort. The insight, experience, guidance, and especially patience were certainly appreciated more than I can describe. I would, also, like to thank Maj Scott Graham, for his involvement. To Dr Gary Lamont, who introduced me to the parallel computing and taught the mathematical foundations underlying that field. To the IT staff of AFIT, who provided speedy and expert assistance and support. Most importantly, I wish to thank my wife for her gentle support and wisdom, and to my daughter, who inspires me every day with her joy and exuberance.

Table of Contents

	Page
Abstract	iv
Acknowledgments.....	v
Table of Contents	vi
List of Figures	viii
List of Tables	x
I. Introduction	1
Problem Statement.....	4
Research Objectives/Questions/Hypotheses	4
Research Focus.....	5
Methodology.....	7
Implications	8
Preview	9
II. Literature Review	10
Artificial Neural Networks	15
Genetic Algorithms	20
Summary.....	28
III. Methodology	29
Complexity of Design.....	29
Focused Heuristics.....	30
Genetic Algorithms for Neural Network Optimization.....	36
Description of pLoGANN	37
Implementation of pLoGANN	44

Stochastic Player	46
Summary.....	46
IV. Analysis and Results.....	49
Results of pLoGANN	49
Results of Random Heuristic.....	55
Results of Random Heuristic vs. pLoGANN	59
Summary.....	62
V. Conclusions and Recommendations	63
Conclusions of Research: What Went Wrong.....	63
Conclusions of Research: What Went Right.....	71
Significance of Research	73
Final Remarks.....	74
Appendix A: Derivation of Parallelization of Fitness Function	75
Appendix B: Theorem: Not all valid Board States of Lines of Action are Reachable	77
Appendix C: Discussion of Offline and Online Reinforcement Learning.....	79
Appendix D: Analysis of Single Elimination versus Round Robin Tournament	82
Bibliography	85

List of Figures

	Page
1: Illustration of Similar Lines of Action Board States	5
2: Mean Squared Error using ANN on 5x5 LoA Board	7
3: Comparison of Some Abstract Strategy Games	12
4: Starting position for Lines of Action	13
5: Perceptron	16
6: A Fully-Connected ANN with a Single Hidden Layer.....	18
7: Single Point Crossover	21
8: Mutation Operator.....	22
9: Column groupings.....	32
10: Diagonal Heuristic	33
11: Quadcounts Heuristic.....	34
12: Nonad Heuristic	35
13: Self/Non-Self Heuristic.....	35
14: Diagram of pLoGANN's Artificial Neural Network and Inputs.....	43
15: pLoGANN Max Performance by Epoch	50
16: pLoGANN Mean Performance by Epoch.....	51
17: pLoGANN Median Performance by Epoch.....	52
18: pLoGANN Mean Performance by Epoch with 5-Smoothing.....	54
19: pLoGANN Median Performance by Epoch with 5-Smoothing.....	55
20: Max Scores by Random Heuristic by Epoch	57

21: pLoGANN Mean Performance Against Random Heuristic	59
22: Example of Unreachable LoA Board State.....	77

List of Tables

	Page
1: Detailed comparison of consecutive epochs.....	52
2: Details of consecutive epochs after 5-smoothing is applied.....	53
3: Results of random heuristic	56
4: Comparison of random heuristic and pLoGANN.....	56
5: Results of random heuristic competing against pLoGANN	60
6: Results of games and matches between random heuristic and pLoGANN	60
7: Results of pLoGANN and RandomANN versus random heuristic	61
8: Results of games and matches between random heuristic and RandomANN.....	61

EVOLUTIONARY ARTIFICIAL NEURAL NETWORK WEIGHT TUNING TO OPTIMIZE DECISION MAKING FOR AN ABSTRACT GAME

I. Introduction

The ability of computers to assess the quality of a single state in a complex search space is a fundamental challenge in the area of artificial intelligence. While it is easy for a computer to quantify elements of a state, such as the number and locations of items A, B, and C, it can be very difficult for the computer to extrapolate how good the state is in terms of reaching a goal. The ultimate goal, perhaps, is to develop a generalized algorithm that can accept any search problem and state description as input, whether a simple tic-tac-toe board or a complex battlefield surveillance photograph, and automatically generate the optimal solution to the problem or a determination that there exists no optimal solution, all without any human intervention. In short, the ultimate goal is to develop a system that produces true artificial intelligence.

Given the current limitations on computational complexity due to present day hardware technology, heuristic algorithms offer much promise compared to other algorithms since they functionally reduce the number of computations needed to perform a search[1]. Thus, unless hardware developments dramatically improve the computational efficiency of computers, the search for a generalized artificial intelligence algorithm is more realistically restricted to a search for a generalized heuristic generation algorithm.

A heuristic algorithm implies a program that detects patterns or features in a particular domain which can be used as a heuristic in a search algorithm [1]. A heuristic

itself is an attribute, feature, or characteristic of a particular problem domain that can be utilized in order to navigate a search algorithm to an acceptable solution within the domain. Consider the problem of efficiently traveling from one corner in downtown Manhattan to a corner in uptown Manhattan. If one wants to walk the shortest distance possible, then one heuristic is compass direction. As one approaches an intersection, compass direction can be used to determine which direction one should turn. This may not lead to the shortest path from source to destination (one could turn into a dead-end alley and be forced to turn around, for example), but it does lead to a good solution under some circumstances. For problems which are so computationally complex that the search for the optimal solution is inefficient or even intractable, a heuristic algorithm can be used to provide a quality solution under many circumstances.

In the pursuit of a generalized heuristic generation algorithm, it is a useful step to focus on a subset of all search problems to identify what may or may not work. Within the pantheon of search problems exists a set known as abstract strategy games (ASG). A game is an ASG if it contains no hidden information and has no stochastic processes intrinsic to the game (determining who goes first in a game is generally extrinsic to the game itself and so does not impact whether or not a game is an ASG) [10]. ASGs in general make excellent fields of study for developing decision making algorithms because they tend to be well-studied and thus have known attributes, they are limited in scope, and they are easy for an average person to understand. Computationally, they can also be quite complex; Go, played on a 19x19 board, has approximately 10^{170} legal game board states, far too many to enumerate by any known method [2]. By comparison, a real-

world environment featuring merely one thousand independent actors (soldiers on a battlefield, for example), each with ten actions they can take within a given timeframe, has a decision complexity of 10^{1000} for a single state, far higher than any commonly played ASG. Thus, while no ASG approaches the complexity of a real world scenario, they are excellent stepping stones towards developing generalized decision making algorithms. After all, if a computer cannot outperform a human in a game of limited scope with perfect information and no stochastic processes, it is unlikely to outperform a human in the dynamic environments faced by humans in everyday life.

This thesis focuses on a single game, the abstract strategy game Lines of Action (LoA, see Chapter 2 for rules) [3], and attempts to develop an algorithm to correctly evaluate the quality of board states. The purpose is quite simple: given a current state of the game, the program evaluates the states produced by every possible legal move available to a player at a given turn. If performed accurately, the state that is rated highest predicts the move that the player should make to maximize his or her probability of winning the game. If all states of the game are correctly evaluated, then the game is strongly solved [4].

Other methods have been used to play not only LoA but other games with varying results. While each method brings both positive and negative attributes to the table, to attempt all of them is well beyond the scope of this thesis. Where appropriate, other methods are compared to the methodology used here, and obviously this project builds upon the knowledge garnered from the efforts of other researchers.

Problem Statement

The goal of this thesis project is to automatically generate a heuristic that advances the ability for computers to play Lines of Action effectively.

Research Objectives/Questions/Hypotheses

This thesis utilizes feedforward artificial neural networks (ANN) to evaluate board states. ANNs have demonstrated both strengths and weaknesses on different types of problems [5]. It generally performs better on small problems, especially since the classic method of training via feed-forward/back-propagation is computationally expensive, so it becomes highly inefficient when training a large structured network. While a game of LoA, with a board size of 64 squares (same as chess and checkers) is significantly smaller than other fields an ANN might examine, such as photographs, a LoA board challenges pattern matching algorithms because nearly identical board states can lead to very different outcomes. In Figure 1, two board states are shown with only a single difference, the location of the dotted white piece. Despite being on the opposite side of the board from where black's potential winning move takes place, the difference in the positions of the white piece is the difference between a black victory and a white victory. This minute difference with drastic consequences can be difficult for a pattern matching algorithm to detect.

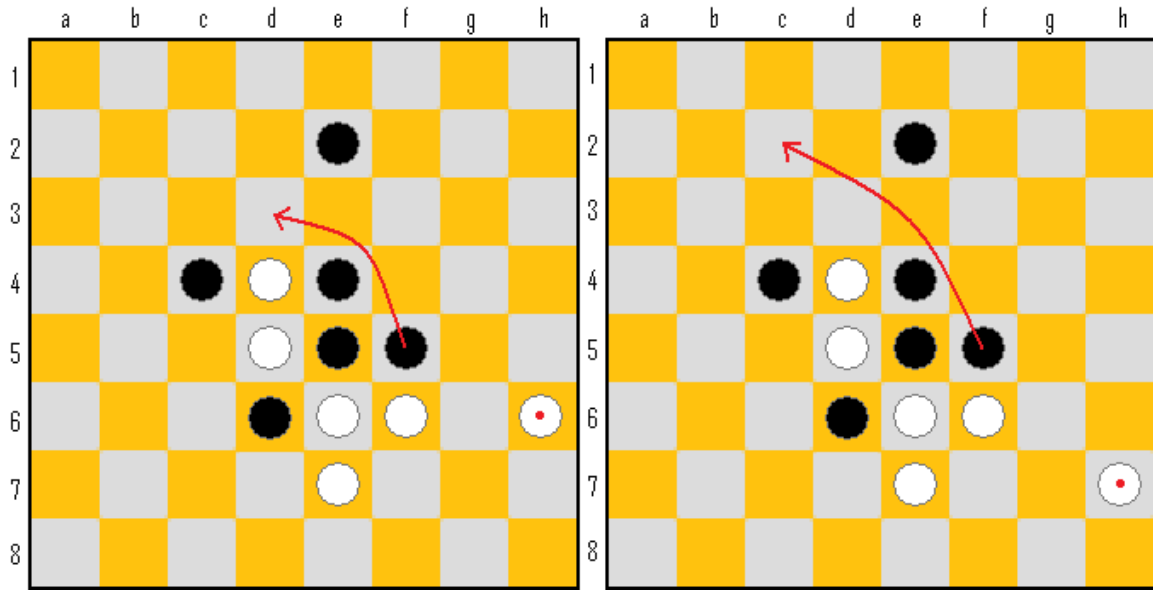


Figure 1: Given that it is black’s turn, black has a winning move in the left board by moving the right-most piece diagonally two spaces to the upper left. In the right board, however, the adjusted position of white’s right-most piece (with red dot) augments black.

Research Focus

The focus is to design and train an artificial neural network to accurately evaluate the board states of a game of LoA. If a program can correctly evaluate every board state, then by always choosing the optimal move it becomes a perfect player. In theory, the closer a program comes to correct evaluations, the closer it comes to solving the game.

One reason ANNs are considered for this thesis is because of past successes ANNs achieved with other games, notably backgammon [6] and checkers [7]. One anecdote of Tesauro’s success using temporal difference learning and ANNs on backgammon occurs as an aside in one researcher’s thesis: “almost every paper listed in the bibliography [of this thesis] cites one of Tesauro’s papers on backgammon” [41].

Aside from ANNs' achievements with other games, experiments focusing on small LoA boards (smaller than the standard 8x8 board), suggest that ANNs have the capability of providing a framework for a quality LoA heuristic. While competing on a 5x5 board, for example, an ANN was able to model the board states of LoA to within a mean squared error of less than 0.02 (see Figure 2). In this experiment a temporal difference learning (TD-learning) [5] hash client competed 10,000 games against a separate heuristic player called ALoA (see Chapter 3 for more details). Each visited board state was recorded in a hash table along with that states value based on the outcome of the game in which that board state was visited. Progress was saved after every 1,000 games to enable analysis. After the completion of all the games, the stored board states and results were transformed into a pattern file for training a simple ANN in JavaNNS. The ANN, consisting of 25 input nodes (one for each cell of the 5x5 board), one hidden layer with 12 nodes, and one output node, was trained using temporal difference learning with three different learning rate values, η . η started at 0.2, but decreased to 0.05 and 0.01 after 100 cycles each. The final results suggest that the ANN is able to correctly model the results of the experiment to within 2% mean squared error. Given this success in modeling a 5x5 board, it is worthwhile to examine whether or not an ANN can model a full 8x8 board.

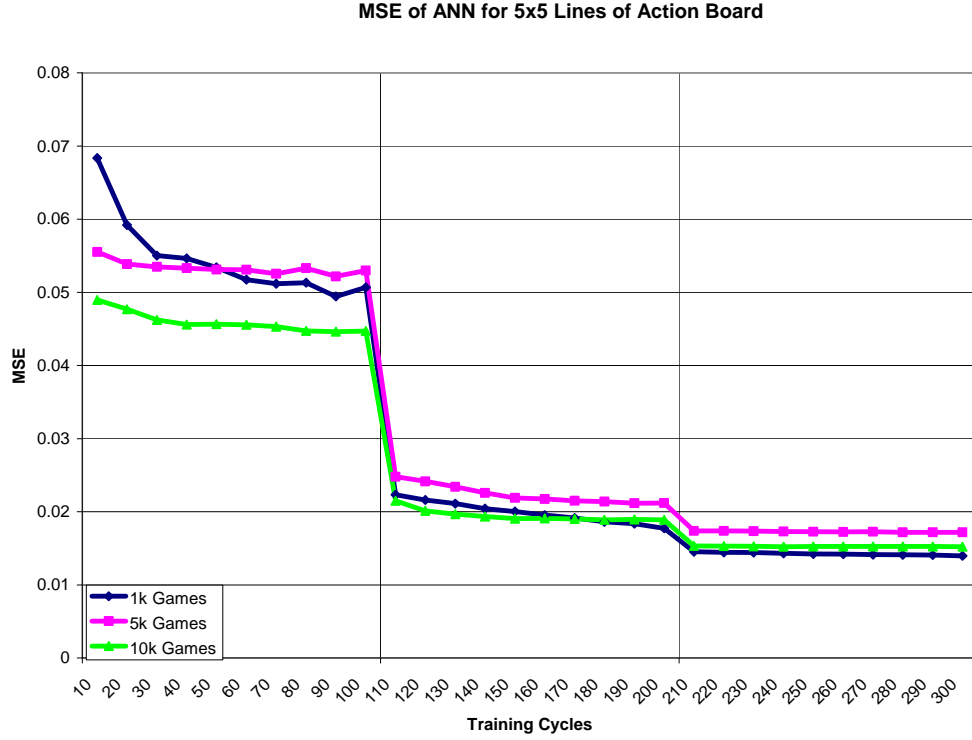


Figure 2: The mean squared error (MSE) of an ANN with 25 inputs, a single hidden layer with 12 nodes, and 1 output modeled the board states that were visited during up to 10,000 games between a random player and ALoA, a heuristic player. As the number of games increased, a greater number of board states were visited, yet the ANN maintained a low MSE regardless of the amount the ANN was trained. The three divisions represent learning rates, η . In order, they are 0.2, 0.05, and 0.01.

Methodology

There are two primary strategies considered in this research effort to develop a LoA player: offline reinforcement learning and online reinforcement learning. The first uses TD-learning [5] and the second uses a genetic algorithm to train the network weights (see Appendix C). Both methods have been used by others in previous research efforts on various domains, including other games, such as backgammon [6], checkers [7], and Go

[27]. Ultimately, online learning is implemented in a program called pLoGANN (Parallel Lines of Action with Genetic Algorithm and Neural Networks). A network structure and a genetic algorithm are designed, with the GA set to train the network in an online environment. Due to the large computational complexity of the online experiment, it is designed to operate over parallel machines and the experiment is conducted on a parallel cluster. The basic methodology for the genetic algorithm is to use tournament competition to select which chromosomes advance to the next generation, where each chromosome represents all of the weights of a network. In other words, each chromosome represents a full artificial LoA player, and the ultimate goal of the second experiment is to evolve a perfect player which effectively solves the game of Lines of Action.

Implications

Lines of Action is an abstract strategy game with a approximately 1.3×10^{24} board states and an average of 30 transitions (moves from one state to another) per board state [17]. As with other such games, such as chess or go, they can be likened to the generalized problem of evaluating a specific state of a domain and determining the best course of action [4]. Finding an algorithm to solve LoA, while perhaps not as prestigious as finding one to solve chess or go, or even the recently solved game of checkers [8], is nonetheless a stepping stone towards discovering a generalized decision-making algorithm with real world implications.

Preview

As detailed in Chapters 4 and 5, the results of these experiments indicated that artificial neural networks are not conducive towards accurately evaluating the board states of Lines of Action. Through a series of experiments, it is clear that ANNs fail to produce a quality LoA player. However, it is beyond the scope of this thesis to prove that ANNs cannot produce a quality LoA player.

Additionally, regardless of the method used, training the network is a time consuming process, which makes ANNs a poor tool to use for changing environments where weights need to be frequently retrained. This is a known property of ANNs, and so only serves as affirmation of other people's findings [5].

A positive outcome of this research effort is the development of a framework, called pLoGANN, for employing a genetic algorithm on a computer cluster. While pLoGANN did not produce positive results on this particular problem, its computational performance and inherent extensibility make it a promising tool for other research efforts. This framework, with modest modifications, could be deployed over the internet and provide a backbone for using a genetic algorithm on an ad hoc network of computers in a manner similar to that used by the @home method of distributed computing [9].

II. Literature Review

This chapter provides an overview of the three primary fields of research of this thesis, reviewing previous research related to this topic and identifying the current state of research in these fields. In order, they are abstract strategy games, including a description of the rules of Lines of Action (LoA), Reinforcement Learning (RL) using Artificial Neural Networks (ANNs), and network optimization using Genetic Algorithms (GAs).

Abstract Strategy Games

An abstract strategy game is defined by Thompson as a perfect information game in which there is no element of chance, and where there is no significant theme to the game which applies meaning to the actions taken in the game [10]. Perfect information implies that the entire board state is known to all players, precluding games in which there is any random element, such as dice or randomized cards or tokens, as well as games in which either player may conceal any state information from his or her opponent, such as in the game Battleship. Such games are always turn-based, in which each player alternates turns. Games in which players move simultaneously, such as with the game Diplomacy, are excluded by this definition.

Sophisticated abstract strategy games have sufficiently large complexity that it is beyond the capability of human beings to know optimal moves for every possible state of the game, thus forcing the player to rely on other resources in order to make a decision on his or her turn. Consequently, the advantage goes to the player who is best able to utilize their resources and counter their opponent's resources.

Allis defines two measures of game complexity: state-space complexity and game-tree complexity [4]. State-space complexity refers to the number of legal positions that are reachable from the starting position. It is important to note that for some games, including Lines of Action, there are legal positions that are not reachable (see Appendix B). Game-tree complexity refers to the total number of leaf nodes in the smallest game-tree which can establish the true value of the starting position. Both state-space and game-tree complexity can be difficult to calculate for some games, so in some cases, upper and lower bounds are calculated instead. Additional notable metrics include the board size and average game length. Board size refers to the number of cells active pieces may be moved to. This does not include off-board collections where pieces may start (as in connect four or Go) or destinations where eliminated pieces are removed to, even, as in chess, if those eliminated pieces may eventually return. Board size nomenclature, however, may be confusing. For example, one can say that both standard chess and standard checkers are played on an eight by eight (8x8) board, despite the implication being that their board sizes are different, since only half cells on a board are legal in checkers. Thus, an 8x8 board in chess implies a 64 cell board size while in checkers it implies a 32 cell board size. Average game-length varies depending on the relative skill of the participants, but a rough estimate is possible by averaging the results of many games played. Figure 3 has shows a comparison of several games and their relative complexities.

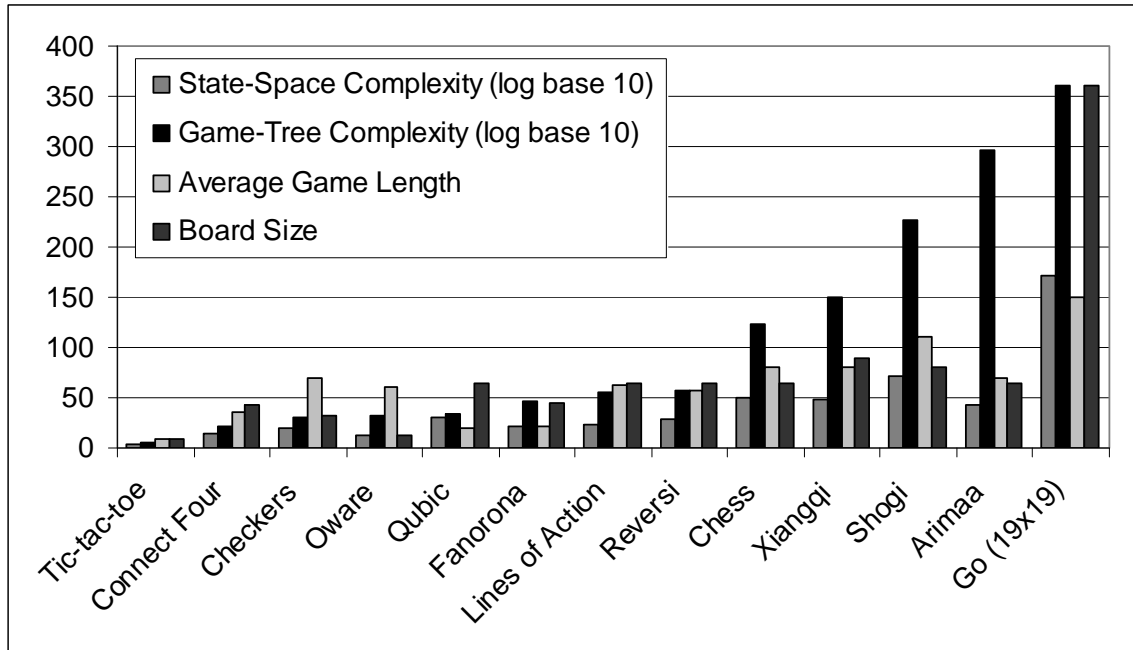


Figure 3: Comparison of Some Abstract Strategy Games [4, 11, 12, 13, 14, 15].

Zero-Sum Property

Another descriptive categorization of games is whether or not they possess the zero-sum property. A zero-sum game holds that the sum of consequences of a single action for all participants must equal zero. In other words, for any action taken by any player, all gains made by players must be offset by equal losses suffered by other players. In laymen's terms, this means that in a zero-sum game, there can be neither win-win situations nor lose-lose situations, but only either win-lose situations or neutral (situations in which no participants gain or lose) situations. Games, especially two-player games, which possess the zero-sum property are perhaps easier to model, since the value of a given game-state from one player's perspective is merely the additive inverse of the value from the other player's perspective [16].

Lines of Action

Lines of Action (LoA) is a zero-sum abstract strategy game developed by Claude Soucie and first published in 1969 [3]. It is played on a regular checkers board which is an eight by eight matrix of alternating black and white tiles. Each player starts with twelve pieces, arranged at the edges of the board as shown in Figure 4.

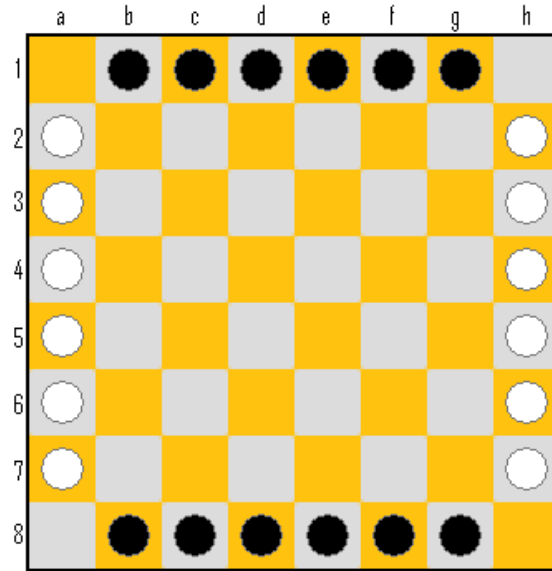


Figure 4: Starting position for Lines of Action.

Traditionally, play begins with black, and then alternates every turn. On each turn, a player may move one piece in any direction. The piece must move a number of tiles equal to the total number of pieces in the line in which the piece is moving, including itself. The moving piece may not land outside of the board. Nor may it jump over a piece of the opposing color, although it is permitted to jump over a piece of the same color. Finally, the moving piece may not land on a tile occupied by a piece of the same color; however, if it lands on a tile occupied by a piece of the opposing color, the opposing

piece is then removed from the game. A side wins when all pieces of that color are connected, which is defined as all pieces forming a single cohesive unit in which each piece is connected to every other piece of the same color via a sequence of orthogonal or diagonal connections [3].

There are approximately 1.3×10^{24} board states in LoA [17]. However, the actual number of achievable board states is somewhat less, due to an unknown number of board positions that are unachievable. The game begins with a branching factor of 36, but averages 30 over the course of the entire game. Winands' study suggests an average game length of 38 turns, but game length varies with the skill of the players. In terms of complexity, as measured by state-space, Lines of Action is comparable to Othello/Reversi. The same is true when measured by game tree complexity.

Due to the extremely high state-space complexity, LoA cannot be solved by enumeration methods. Furthermore, LoA is currently not solved under any of Allis' definitions of solved abstract strategy games [4]:

1. *Ultra weakly solved*: the outcome of perfect play from the initial position is known. However, this does not imply that the perfect playing strategy itself is known.
2. *Weakly solved*: perfect playing strategy from the initial position is known.
3. *Strongly solved*: perfect playing strategy is known for all legal positions.

Van den Herik, et al., conjecture that LoA will be weakly solved by 2010 [18], but at the time of this paper’s completion, this has not occurred.

Lines of Action presents a different type of challenge from other well-known abstract strategy games. While mathematically comparable to Othello/Reversi [4], the play of LoA, as well as its unusual end-game conditions, creates a state space where nearly identical positions can lead to completely opposite outcomes. While this is plausibly true of any ASG, especially an ASG known to be strongly solved, LoA is notable because of the frequency of such states. As there is no clearly defined ‘side’ of the board, as there is with chess or checkers, and because pieces can move across the board in irregular ways, the most minor of differences between two board states can be highly consequential.

To date there are several very successful artificial intelligence programs designed to compete against humans in Lines of Action. The most notable is MIA by Mark Winands [17] which has won several Computer Olympiad events. A couple of other strong programs are YL and MONA, both developed by Billings and Björnsson, which use very different approaches [32].

Artificial Neural Networks

An Artificial Neural Network (ANN) is a computational model derived from observations of biological neural networks. First introduced by McCulloch and Pitt in 1943 as a system of propositional logic, ANNs simulate a network of synapses, where each artificial synapse represents a logic gate. McCulloch and Pitt demonstrated that a network of so-called MP-Neurons could express any statement in propositional logic

[19]. However, neither McCulloch nor Pitts were interested in artificial intelligence, and so it was others who adapted their model towards the purpose of computer learning. Since then, the basic ANN has developed into a network of units called perceptrons that are typically organized into layers and feature dynamic properties that are altered by training.

Perceptrons

In 1958, Frank Rosenblatt developed the perceptron, which replaced the simple MP-Neurons as the building blocks of a ANN [20]. A perceptron, like the MP-Neuron, is modeled after a single synapse. However, unlike its predecessor, the perceptron is not strictly a logic gate; rather, as shown in Figure 5, it represents a function, with n weighted inputs, a weighted *bias* input, an *activation function* g , and a series of output links that may connect to other perceptrons in the network.

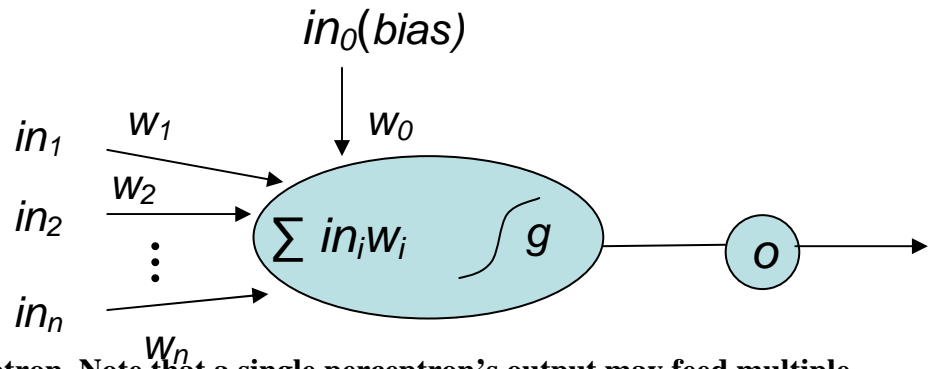


Figure 5: A Perceptron. Note that a single perceptron's output may feed multiple destinations, such as multiple perceptrons in a subsequent layer.

The perceptron functions by taking the summation of all the weighted inputs and feeding it to the activation function g . The function activation may take many forms, notably threshold or a sigmoid functions [21]. It is important that g not be a linear

function, however, as that would allow a network of such perceptrons to be reduced to a linear function, thus negating the purpose of having a network in the first place [21].

A threshold function would produce a discrete set of outputs, such as $\{0, 1\}$. In such a simple case, the perceptron could act as a logic gate, given appropriate inputs. The more general perceptron uses a sigmoid activation function to return a value within a finite range, such as between -1 and 1. It is called an activation function because one extreme of the output range is considered active while the other inactive, and so the perceptron is only ‘triggered’ when the sum of the weighted inputs exceeds some threshold. The bias input defines the threshold, and results in the perceptron being activated only when the sum of the weighted inputs exceeds the weighted bias.

Perceptron Networks and Layers

A single perceptron is limited in its ability to separate inputs. A threshold perceptron is a linear separator, because it determines a straight line (or n -dimensional plane) through the weight space and activates on all inputs on one side of the line and none of the inputs on the other [21]. However, such a perceptron by itself cannot distinguish between inputs which are not linearly separated. For such problems, a network of perceptrons is needed.

A neural network consists of multiple perceptrons that are connected together in some arrangement. The simplest such arrangement is called a perceptron layer, and consists of a set of perceptrons arranged parallel to one another where they each are fed the same inputs and produce independent outputs. In a single layer network, the number of outputs equals the number of perceptrons. In a standard multi-layer network, the

number of outputs equals the number of perceptrons in the outermost, layer, called the output layer. Networks that do not fit this model are called convoluted neural networks, and may be organized any number of ways [24].

The layer that is connected directly to the inputs is called the input layer. Any layers between the input layer and the output layer are called hidden layers. Networks may be fully-connected or partially-connected. A fully connected graph implies that every input feeds into every perceptron in the input layer and that every perceptron in each layer provides input for every perceptron in the subsequent layer (see Figure 6). Hidden layers are important because they add flexibility in terms of what inputs activate the network as a whole. With sufficient hidden inputs it is possible to represent any continuous function [21].

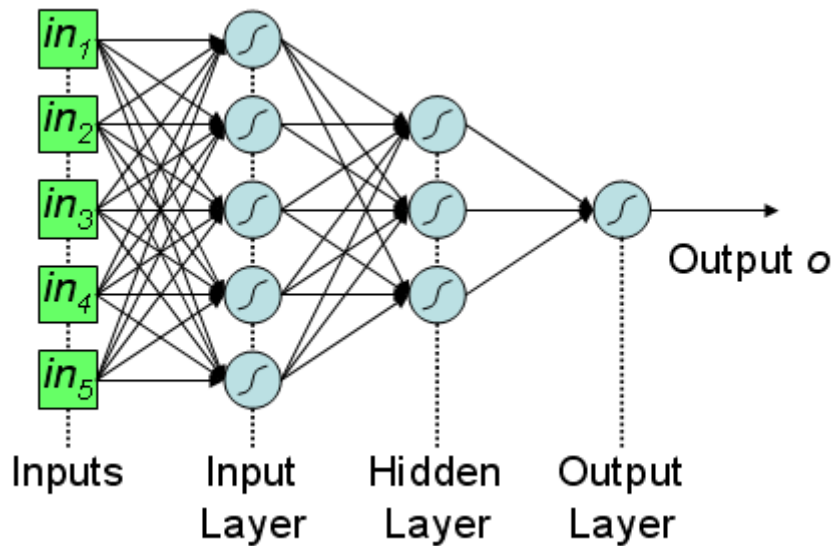


Figure 6: A fully-connected ANN with a single hidden layer.

Performance is a concern when it comes to NNs. On a fully connected graph, the number of connections equals the sum of the products of the size of each layer and it's

subsequent layer. Thus, for the ANN in Figure 6, the layers (including the inputs) are of size, 5, 5, 3, and 1. Thus, the total number of connections is $(5*5) + (5*3) + (3*1) = 43$. Computing the output of a ANN is of complexity $O(cg)$, where c is the number of connections on the graph, and g is the cost of the activation function). Since training a ANN involves re-computing the network many times, the operational efficiency of the network can degrade very quickly as the size of the network increases.

Back Propagation and Direct Weight Training

The two methods used to train the ANN in this thesis are back propagation and direct weight training. Back propagation is performed by experimentally measuring the delta, or difference, between the ANN's evaluation of a state's value and the state's actual value. The delta is then used to incrementally adjust the weights of the network, beginning with the output layer, and working backwards towards the input layers [21].

Direct weight training, on the other hand, utilizes a methodology which directly sets some or all of the weights to specific values. A number of methods for setting the values exist, including genetic algorithms [1], ant colony optimization [40], and simulated annealing [1]. The strategy behind direct weight training is to determine the values of the weights that will optimize the network without dealing with the negatives of back propagation, which include lengthy computation times and the risk of getting stuck in a local minima.

Neural Network as a Continuous Optimization Problem

Another way to view the problem of optimizing the weights of a network is to examine the problem globally. Given a network with n weights, there exists a n -

dimension space consisting of all possible permutations of weights. If the goal is to find the single permutation that yields the optimal performance of the network, the problem of training the network becomes an optimization problem. When the search space for an optimization problem is so large that a full search is infeasible, other methods can be used to search for the optimal solution. One such method is a genetic algorithm.

Genetic Algorithms

Genetic Algorithms (GAs) were first introduced by John Holland in the 1970's [22]. Modeled after cellular mechanics, a GA is a stochastic process designed to solve combinatorial optimization problems. A GA maintains a set of values, called *alleles*, that form a *chromosome*. Each chromosome exists in genome space, and represents a solution space the GA is operating on. A GA uses a stochastic process to evolve a population of initially randomly generated chromosomes into better solutions. The three primary processes used by a GA are crossover, mutation, and selection. The domain over which these processes operate, however, depend entirely on how information is encoded into the chromosomes.

Crossover

Crossover, also known as recombination, is the basic act of producing new chromosomes. Analogous to sexual reproduction, it does this by taking two chromosomes and swapping a set of alleles between the two chromosomes, thus producing two additional chromosomes. Different implementations of the crossover operator use varying techniques, but one of the simplest methods is the k -point crossover [22], where k crossover points are selected at equal points in the two chromosomes. Thus, each

chromosome is divided into $k + 1$ chromosomal sections, which are then swapped with one another. In the simplest case, k equals one, such that each chromosome is split in two and one of the sections is swapped between the parents. On the other side of the spectrum is uniform crossover, where a crossover point is placed between any two alleles with equal probability throughout the length of the chromosome [23].

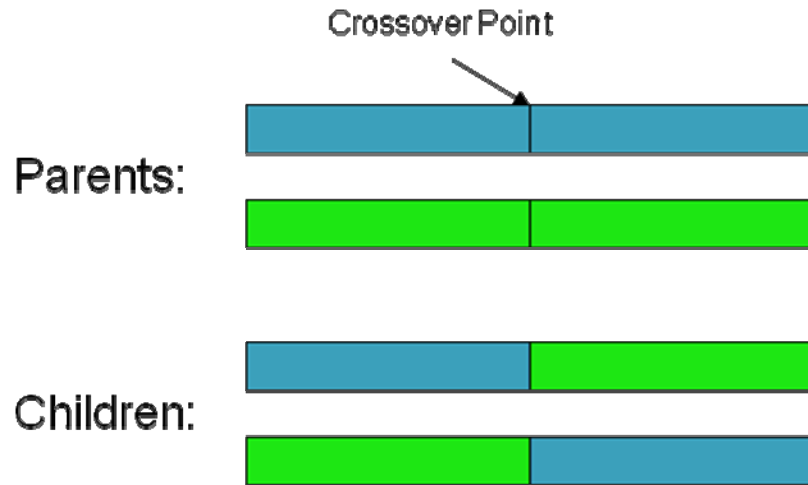


Figure 7: Single Point Crossover.

Mutation

Mutation is the process where a portion of a solution is randomly changed. Whereas the basic crossover operator uses two parents, the mutation operator uses only one, and is thus analogous to asexual reproduction. With this operator, a chromosome is selected, and then a random number of alleles are changed randomly. If an allele is represented by a single bit, then the change is simply a *not* operation on the bit. As with crossover, there are many ways to implement mutation. There is a probability p_1 that a given chromosome is selected for mutation, and then within that chromosome a probability p_2 that a mutation operator acts on a given allele. If p_1 equals one, then the

mutation operator will proceed over every chromosome in the entire population, mutating each allele with probability p_2 . Depending on the design of the GA, the mutated chromosome may replace the original, or just join the population alongside the original.

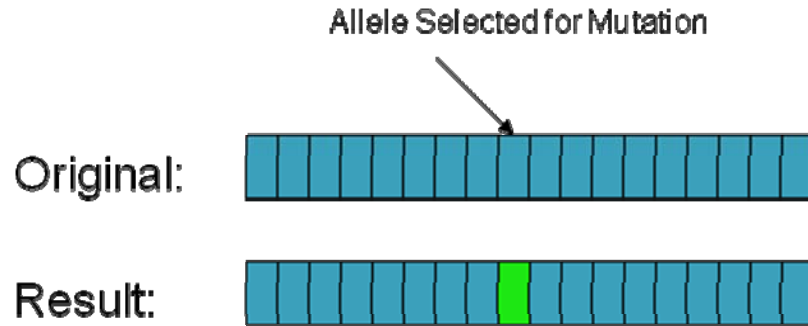


Figure 8: Mutation Operator.

Selection

Selection is used to trim the size of the population in each generation of the genetic algorithm. Although there are many variants of the selection operator, the basic principle is that the operator selects n chromosomes from the existing population to be used for crossover and/or mutation, and discards the rest, thus leaving the population of the next generation at size n . The operator uses a fitness function to determine the value of each member of the population, and based on those evaluations determines which chromosomes advance [31].

Fitness functions take various forms, and the function itself is dependent upon the domain. For some domains, the fitness function measures exactly the value of the solution represented by each chromosome, whereas for other domains, an exact value cannot be determined, and so the fitness function may only represent an approximation as

to what value the solution has. An example of the former is a fitness function for an allele that represents a Hamiltonian circuit, such as a sequence of cities used to solve the Travelling Salesman Problem [25]. In this case, the fitness function can accurately measure the length of the circuit; however, it cannot necessarily determine if the chromosome represents an optimal solution.

Even if it is possible to determine a correct value of a chromosome, the developer of a GA may opt to use another fitness function that only approximates the value in order to improve performance. Because GAs run over many generations and the fitness function is performed over each member of the population (in most implementations of GAs), a slow fitness function may introduce a bottleneck in the algorithm. Thus, it is important to develop efficient methods of evaluating the population of chromosomes each generation.

In most GAs, fitness functions operate over all chromosomes, or sets of chromosomes, independently. Since the chromosomes can be evaluated independently (or in the case of tournaments, in small, independent, subsets), these functions lend themselves easily to parallelization. Based on Gustafson's Law [26], given a computer or network with P processors, and a population with n chromosomes, parallelizing a fitness function with efficiency $O(x)$, which operates over every chromosome individually, improves the performance of the fitness function from $O(xn)$ to $O(xn/P + Pt_c)$, where Pt_c is the cost of communication to the different nodes. When the cost of communication is very low, this yields a near linear improvement in performance over a single-node implementation (see Appendix A).

Once evaluated, the selection algorithm determines which chromosomes to advance to the next generation, and which to discard. A simple elitist algorithm may simply select the n highest performing chromosomes, whereas a stochastic algorithm might apply a probability of selection for each chromosome based on their fitness function, such that the best performing chromosomes have a higher probability of being retained. In certain domains, very poorly performing chromosomes may yet be adjacent to very well performing locations in the solution space. In such cases, it is said to contain good ‘building blocks’ [28], and a GA can use those building blocks to find the better performing neighbors. Thus, strictly elitist methods sometimes lose opportunities to explore possible new promising avenues.

Encoding

The singular requirement of a GA’s encoding algorithm is that an entire solution must be encoded into each chromosome. The method used is problem-specific, as it depends on what kind of data is to be encoded. As an example, real values, such as 32-bit integers can be stored as entire values (32-bit alleles) or as individual bits (thirty two 1-bit alleles). The former is susceptible to change only by mutation, while the latter is also influenced by crossover, since the crossover point could be placed in the middle of the bits that make up that integer value. Advocates for either approach exist [25]. One stated advantage for the binary approach is that it maximizes the number of alleles, and thus maximizes the capabilities of the crossover operator [28]. However, no proof exists that this is the best approach, and in fact there are examples of real-valued encoding methods outperforming the binary approach on benchmark problems [23].

Schema Theorem

The idea of the Schema Theorem is that in each generation of a GA, there are schemata within the population, and the proportion of the population of one generation that has a particular schema is a function of the proportion of the previous generation that had that schema and its fitness value, independently of other schema in the population.

A schema is “a subset of the space A^l in which all the strings share a particular set of defined values” [29]. To draw a parallel from real life, an example of a schema might be the gene that controls eye color. This gene is a subset of all genes in which people with the same (or similar, at least) colored eyes share certain values in their DNA. The Schema Theorem then argues that the proportion of people in the next generation that have a particular eye color is a function of the proportion of the current generation that have that eye color and the fitness value (undefined as that is for humans) of that eye color. A key assumption of the Schema Theorem is that the fitness of a particular schema is the average fitness of members of the population that have that schema.

Returning to GAs, two important properties of schemas are needed for the theorem. The first is the length of a schema. This is simply the number of nodes between the first and last defined positions of a schema, inclusive. Thus, the length of “1**4*”, where “*” represents an undefined position, is four. Length is important because of the nature of crossover. The longer a schema is, the more likely it will be split during crossover. For example, if the length of a schema is one (“*2*”), it cannot be split, and is guaranteed to be preserved during any crossover operation. On the other hand, if the length is equal to the string length (“123”), any non-trivial crossover operation will split

the schema, and it will be lost to the next generation unless the other ‘parent’ also happens to have that schema, or enough of the schema that crossover preserves it. Similarly, two parents without a schema could produce an offspring with that schema by coincidence.

The second important property is order. A schema’s order is the number of defined positions of a schema. Thus, the order of “1**4*” is two. Order is more important when mutation is taken into consideration, as the greater a schema’s order is, the more likely it is to be mutated out of a given generation, and the less likely it is to be randomly produced by mutation. For example, on a schema of order one on a binary string, if mutation occurs on that single defined position, it will wipe it out. However, if mutation occurs on that defined position on a member that does not have that schema, mutation may create that schema in that member.

While the ability to predict the proportion of a population that carries a particular schema from one generation to the next is nice, it has failed in explaining the performance of GAs in general, which was one of the hopes Holland had when he first proposed with the Schema Theorem. Criticisms of Schema Theorem have focused mostly on the fact that although it is likely true (nobody has definitively disproved the mathematics behind the Schema Theorem), it is not relevant to long-term performance [30]. There is a lot to be said about the assumption that a schema’s fitness value is simply the average value of members of the population that happen to have that schema. After all, this makes the fitness value more dependant upon other members of the population than on the actual amount that a particular allele contributes. Using a real world example,

the Schema Theorem, applied to humans, implies that the fitness value of the gene that causes humans to grow wisdom teeth or the appendix is a function of the average fitness value of people, which is a debatable proposition.

The Schema Theorem was an early attempt to predict the behavior of a genetic algorithm. While it may do so from one particular generation to the next, large numbers of critics have shown that it is not sufficient in explaining long-term behavior. At the least, the Schema Theorem provides important background knowledge and an introduction to the rather imprecise stochastic math, filled with assumptions, that plays an important role in GAs.

Trends

Genetic Algorithms operate over large, sometimes ill-defined domains over the course of many generations. A developer must design a GA around competing interests, namely *exploration* and *exploitation* [25]. In order to improve their odds of finding optimal solutions, GAs must explore as wide a breadth of the solution space as possible. However, they must also hone in on good solutions once a promising sector of the solution space has been discovered. In short, a good GA explores the space sufficiently, but also converges once an optimal solution is found. The two basic reproductive operators, crossover and mutation, are designed specifically to meet the exploration goal, while the selection strategy for the new population and the population members used in crossover meets the exploitation goal.

As illustrated by the Schema Theorem, crossover seeks to improve the performance of a population by replicating good schema. Thus, crossover is designed to

exploit good traits. Mutation, on the other hand, being completely arbitrary, is more suited for exploring the solution space. In particular, mutation can help a GA to escape local minima by randomly moving elsewhere in the solution space. If the new found chromosome performs better than the rest of the population, it should propagate into further generations. By tweaking mutation rates, a designer can also change the performance of a GA midway [28].

Fitness and selection also play a key role in the balance between exploration and exploitation. An elitist selection algorithm is more prone to converging. However, as in a basic hill-climbing algorithm, premature convergence may lead to a local minimum, perhaps even a very weak local minimum, rather than the global minimum. On the other hand, non-elitist methods may not converge at all, or even worse, may even discard the global minimum. For this reason, it is wise to retain copies of the highest scoring chromosomes, even if they are discarded by the selection algorithm.

Summary

The game Lines of Action has a very quiet global following. However, within the field of artificial intelligence it presents a challenging environment for any pattern recognition algorithm, due to the symmetry and movement within the game. In order to overcome the prohibitive search space presented by this design, modern sophisticated heuristics, including genetic algorithms, are utilized to attempt to find good, though not necessarily optimal, solutions.

III. Methodology

This chapter describes the design and implementation of pLoGANN, a distributed program designed to learn to play Lines of Action. It includes a discussion of some of the focused heuristics that can be used to analyze a LoA board, one of which is prominently featured in pLoGANN. Finally, this chapter describes the process of testing pLoGANN as well as a separate program to assist with the analysis of pLoGANN's performance.

Complexity of Design

A significant component in designing an experiment using an artificial neural network is that one must identify the structure of the network. Identifying the optimal network structure for a given problem domain is itself, however, a combinatorial optimization problem [33]. To that end, one must first examine the domain with the hope that patterns or clues can be detected and then perhaps exploited. A simple example is the game tic-tac-toe, where after a cursory examination, it is clear that the center square is highly important when playing the game.

Lines of Action is played on an n by n board consisting of n^2 tiles, each of which may be occupied by a white piece, a black piece, or an empty space. Thus, there is a maximum of 3^{n^2} possible board states. However, not all such board states are valid board states in LoA due to the restrictions of pieces. For example, it is not possible for a board state to exist with $2n$ tiles occupied by black pieces since there are never more than $2n - 4$ black pieces in any game of LoA. For an 8x8 board, the ceiling of the number of legal positions has been calculated to be approximately 1.3×10^{24} [17]. However, this is just a

ceiling, as there are some states that are unreachable given the standard starting position of LoA (Appendix B).

The task of the ANN then is to accept as input 3^m possible board states and provide meaningful evaluations, where board states that indicate an approach to a loss have a lower limit of -1 and board states that lead to victories approach an upper limit of 1 . Unfortunately, there is no upper bound to the number of topologies that can be used for a given network. However, larger networks take longer to compute, and in a simulation that requires millions of board evaluations, the cost of using a larger network becomes computationally expensive. In general, there are two prudent approaches to selecting a network topology. The first is to design a fixed topology based on best practices, although it amounts to little more than a best guess. The second approach is to simultaneously evolve the weights of the network and its topology. The latter approach has produced some promising results. Stanley and Miikkulainen [34] developed a routine called NeuroEvolution of Augmenting Topologies (NEAT) that performed well on several problems. Unfortunately, evolving an optimal topology is perhaps even more computationally difficult than it is to optimize the weights of a fixed topology. Furthermore, to attempt such a feat is tantamount to changing the scope of the original problem. Consequently, a fixed topology is used for these experiments.

Focused Heuristics

As part of these experiments, different elements, or sections, of the board state are evaluated separately by the network, under the hypothesis that focusing extra attention on certain aspects of the board may yield better results. Within the ANN, some elements are

represented merely by a low level sub-tree which accepts as input each cell within that element, while other elements are fed through an interpretation algorithm before being passed to the ANN. Regardless, each is a focused heuristic that augments the overall interpretation of the board by the network. Borrowing from the heuristics used by Winands [35], these are (number of inputs reference an $n \times n$ sized board):

Board: Each cell of the board is fed directly into an input. This requires n^2 inputs, where the values of the inputs are numerical representations of the cell. In these experiments, if the cell is occupied by one's own piece ("self"), it is represented with a 1, while a cell occupied by the opponent's piece is represented as -1 and an empty cell is represented as 0.

Rows and Columns: Each cell of each row and each column are grouped together. Thus, there are n rows, each with n inputs, and the same for columns (as demonstrated in Figure 9), for a total of $2n^2$ inputs grouped into $2n$ sets of n individual cells. The input for each row and column is the number of pieces (both self and opponent) in that row or column.

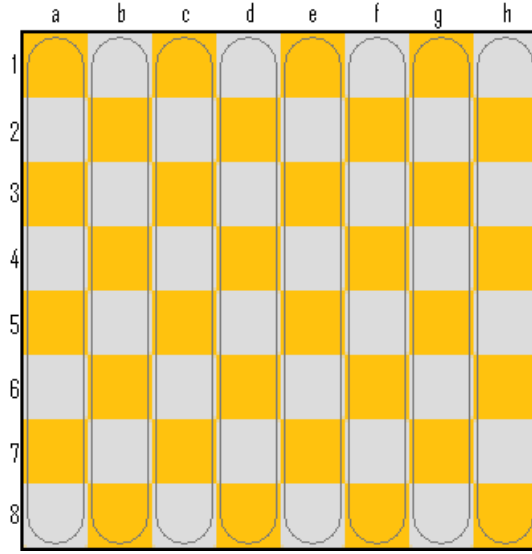


Figure 9: Column groupings.

Diagonals: Diagonal segments are grouped together. Each segment ranges in length from one cell to n cells, and for each diagonal direction, there is one segment of length n and two segments each of lengths 1 through $n-1$, for a total of n^2 inputs per direction, or $2n^2$ inputs total. The two diagonal directions are forward leaning and backward leaning, or, if referring to cardinal directions, NW-SE (shown in Figure 10) and NE-SW. The input for each of these segments is then the number of pieces, both self and opponent, on each diagonal.

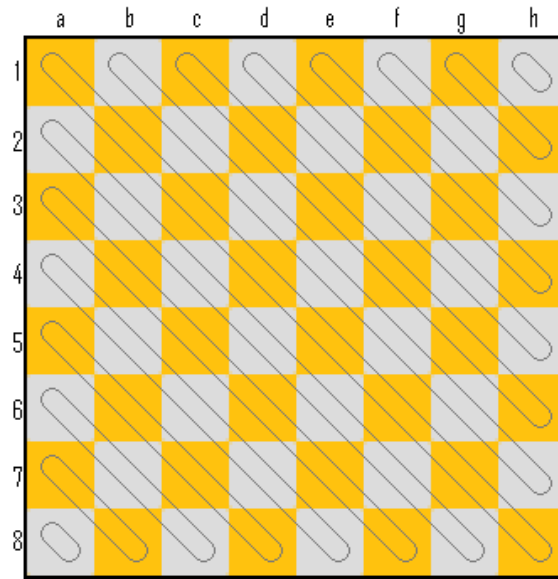


Figure 10: NW-SE Diagonal groupings.

Quadcounts: Quadcounts refer to a heuristic developed by Winands [36]. Each quad represents an overlapping 2x2 section of the board. So that each space on the board is counted equally, quadcounts assume the existence of a phantom border to the board with a depth of one cell in all directions, and are always empty. Thus, each cell of the board appears in four different quads, once in each position within a quad. Quadcounts simply maintain the number of each type of quad that a side has. There are six possibilities, as illustrated in Figure 11:

- 0: the side has no pieces in the quad
- 1: the side has one piece in the quad
- 2: the side has two pieces in the quad, arranged orthogonally
- 3: the side has three pieces in the quad
- 4: the side has four pieces in the quad
- 5: the side has two pieces in the quad, arranged diagonally

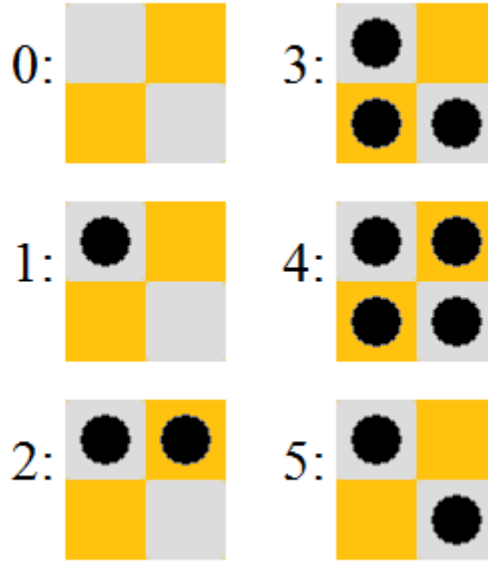


Figure 11: Examples of quadcounts. Each quad can appear in any orientation.

Quads are treated as combinations, rather than permutations. This allows quads of the same shape to be treated equally regardless of orientation. The quadcounts, then, are simply the sum of all quads of a particular type. The network accepts as input the count of each type of quad, for a total of 6 inputs.

Nonads: Nonads represent a sequence of samplings of the board state. Each nonad is a 3x3 square of the board. Unlike Winands' quads, nonads do not assume a phantom border. Thus, there are $(n-2)^2$ nonads for an $n \times n$ board. Each nonad is evaluated by assigning a point total for every piece of the active side in the 3x3 area, based on where in the 3x3 area that piece is located. The points are awarded on three schedules (highlighted in Figure 12): 1.0 for the center cell, 0.8 for each side cell, and 0.75 for each

corner cell. As with quads, the opponent's pieces are ignored. The total values of each nonad are fed into the network, requiring $(n-2)^2$ inputs.

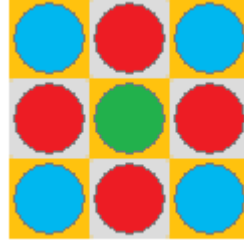


Figure 12: A nonad with the center schedule in green, the side schedule in red, and the corner schedule in blue.

Self/non-self: Self and non-self boards are full representations of the board in which one side is completely ignored. For self, the input of each cell can hold two values: one if the cell is occupied by a piece of the active player's color, and the other if the cell is empty or occupied by a piece of the opponent's color. As shown in Figure 13, the opposite holds true for the non-self evaluation. Both self and non-self representations use one input per cell of the board. Thus, these representations require a total of $2n^2$ inputs.

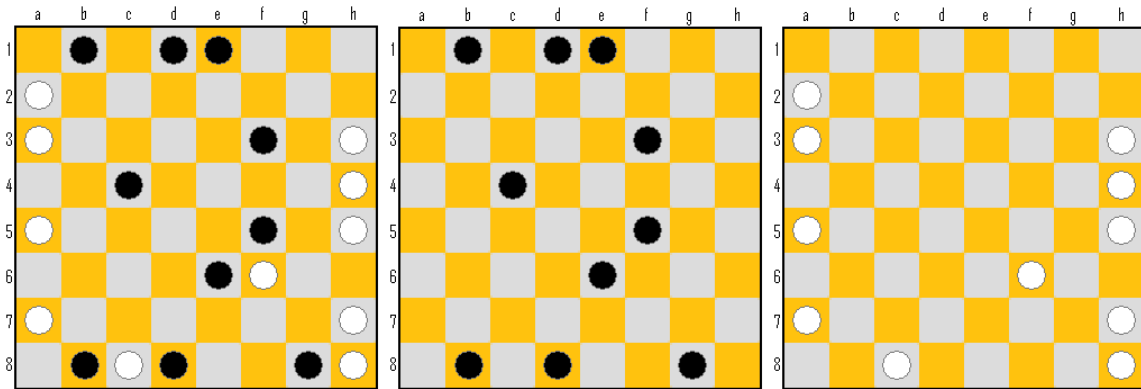


Figure 13: The self/non-self heuristic takes a regular board (left) and separately evaluates the self (center) and non-self (right) components.

Genetic Algorithms for Neural Network Optimization

The approach used to train a neural network to play LoA is to use a genetic algorithm to evolve the weights of the neural network [33]. In contrast with offline reinforcement learning, which was considered (see Appendix C), this requires the network to learn as it plays in an online environment. In this GA, each chromosome represents all of the weights of a network, where each allele is a particular weight. Each generation features both crossover and mutation to increase the size of the population. Tournament selection [31] is then used to determine which chromosomes persist to the next generation. The tournament is elitist, which guarantees that if there is a single best chromosome in the search space, that once found, it cannot be eliminated from the population. In this case, the single best chromosome would result in a network that acts as a perfect player, effectively solving the game. Exploration is achieved via mutation, and exploitation via crossover.

For Lines of Action, the search space is very large. Each allele represents a single weight. Each weight is stored as a double, and ranges from -1 to 1. Since there are approximately $6.9 * 10^{18}$ values between -1 and 1 that can be stored as a double, there are $x = 6.9 * 10^{18}$ possible values for each weight. For a given neural network, there are w weights. Thus, the GA must contend with a search space of x^w possible network configurations. In my experiments, each network has 2041 weights, producing a total search space of size $1.531 * 10^{38,451}$.

Tournament selection is conducted by having two chromosomes compete in a match against one another. Each chromosome is used to seed the weights of a neural

network. The two networks then play two games, swapping colors after the first game. During each turn of these games, the network evaluates all the possible moves, which consists of the resultant board state of each move being evaluated by the network. Each evaluation results in a score between -1 and 1. Whichever move has the highest score is selected. The winner of the match is determined by which network wins the most games. If they each win one game, then a tie-breaker is used to determine the winner. The tie-breaker awards the victory to whichever network achieved its win in the fewest number of moves. If each side won its game in an equal number of moves, or if both games resulted in a draw, then one chromosome is selected at random as the winner. The tournament is single elimination; once a chromosome has lost a match, it is no longer considered for advancement to the next generation. Consequently, given a pool of size p and a selection limit of size s chromosomes that are to be selected for advancement, the tournament selector must play $p - s$ matches, or $2(p - s)$ total games per generation.

Description of pLoGANN

To perform the online learning experiment, a program is developed called Parallel Lines of Action with Genetic Algorithm and Neural Networks, or pLoGANN for short. Written in Java, it is a network based program with two parts, the master and the slaves. The master controls the course of the genetic algorithm, while the slaves simulate games of Lines of Action and transmit to the master the results.

The master performs three primary tasks. First, it manages the genetic algorithm, which includes performing the genetic operators, handling the loading and saving operations, and managing the populations. Second, it commands the slaves, to include the

network communications protocols as well as assigning the slaves tasks. Finally, the master performs evaluations every ten generations and to enable the experimenter to monitor the progress of the algorithms evolution.

The slaves on the other hand have more straightforward responsibilities. It need only establish communications with the master and then play games of Lines of Action. The master sends the slave two chromosomes, A and B , which consist of a set of weights in a known order. The slaves apply those weights to two identically structured pre-defined artificial neural networks, and then use those networks as the players in a two game match of LoA. Each chromosome plays one game as black and one as white. The winner is determined using the following criteria:

1. If a chromosome wins both games
2. If a chromosome wins one game and ties the other
3. If a chromosome wins one game, loses the other, but the win occurred in fewer turns than the loss
4. If a chromosome wins one game, loses the other, and games are of equal length; or if the chromosomes tie both games, then chromosome A is assigned as the winner (note: a game is declared tied in pLoGANN if, after one hundred moves, neither side has achieved victory)

The purpose of the tiebreaker rules 3 and 4 are twofold. Rule three favors a strong and fast offense, while rule 4 favors longevity, which results from the fact that chromosome A is passed down from the previous generation, while chromosome B is newly evolved. Thus, a form of elitism is established, which says that a chromosome that

has won a match persists until it is definitively defeated. As a caveat, in the inaugural generation, where both chromosomes *A* and *B* are newly created, rule 4 implies that *A* is selected because it was randomly created first.

The organization of the genetic algorithm is as follows. Each generation begins with a population equal to the number of slaves connected to the server. In pLoGANN, this number is defined at compile time, and is based on the known number of computer nodes available to the experimenters. For the final experiment, performed on a cluster computer with 32 dual-core processors, the master operates alone on a single processor while each of the other 31 processors are assigned two slaves, for a total of 62. Thus each generation begins with 62 chromosomes.

The first step in each generation is to expand the population by using genetic operators. Each individual produces two offspring via mutation and one offspring via recombination with another parent. For each individual, another individual is randomly selected, which means that on average, each chromosome will parent two offspring with different partners through recombination. pLoGANN uses single-point crossover for recombination. Once the crossover point is randomly determined, the alleles to the left of the point from parent *A* are spliced to the alleles to the right of the crossover point from parent *B* to form the offspring. The number of newly created chromosomes equals the number of chromosomes that existed at the start of the cycle, which equals the number of slaves. Thus, at this point there are four individuals in the population for each slave, producing a total of 248 participants each generation.

The mutation operator iterates twice over each chromosome. During each pass, it changes any given allele with probability 0.15. If an allele is selected to be mutated, its value is simply replaced with another uniformly distributed random floating point value within the range of -1 and 1. Each pass creates a new mutated offspring. Given that each chromosome has 2041 alleles, each mutated offspring has on average 306 alleles that differ from its parent. This is a high mutation rate, but is deemed necessary to ensure that pLoGANN explores the solution space as much as possible, and is meant to compensate for the elitist tournament selector, which may induce premature convergence. In the absence of any deterministic way to determine an optimal mutation rate, 0.15 was selected basically because it is a round number that is high enough to produce large mutations but still low enough that it does not mutate the majority of the alleles of a given chromosome with any great frequency.

Tournament selection is used to once again reduce the population. Four chromosomes, one old and two new mutated individual, and one bred (via crossover) individuals, all randomly selected, are grouped together and sent to each slave. They compete in a “final four” style tournament consisting of a total of three matches, after which the slave returns to the master the identity of the winner. Elitism is employed in order to maximize the number of solutions examined over the course of the experiment. Thus, once a solution has been defeated, it is permanently removed from the population to make room for as many offspring as possible. The master retains the winner and discards the losers. Once all slaves have returned to the master the winners of their

tournaments, the generational cycle is complete, and reproduction takes place again to begin the next cycle.

Every ten generations, the master then performs two tasks. First, it competes each of the 62 surviving members of the latest generation against another artificial LoA player, called ALoA (AFIT LoA) in this thesis. This is also done in parallel, with each slave performing the evaluation for one chromosome. Each evaluation consists a two game match against ALoA, after which the slave returns the score of each game to the master. Second, the master saves all the chromosomes from that generation in order to allow the experimenter to recreate any ANN that shows promising results. By competing the entire population against a strong artificial player and recording the results, a measurement is generated that reveals to what extent the population is evolving into a stronger pool of players. This measurement is not used by the GA, but rather is only a tool for monitoring the performance and progress of the GA extrinsically.

ALoA uses several heuristics to evaluate board states. The two primary heuristics (other than connectivity, which is a terminal state anyway) are centrality and quadcounts. Centrality measures the average distance between all of the pieces of each side. The shorter the distance between all of one's own pieces the better. Furthermore, an additional penalty is levied against pieces that are on the edge of the board, since they are not only less mobile but are also easier to block. Quadcounts are evaluated as described above. If a side has either three or four pieces in a quad (quads 4 and 5 as shown in Figure 11) and that quad is located in the proximity of the center of all of that side's pieces, the board is evaluated more favorably. Finally, taking advantage of the fact that

LoA is a zero-sum game, the opponent's pieces are evaluated using the same heuristics, and the final evaluation is the difference between the evaluations of one's own pieces and one's opponent's pieces. ALoA, which also uses a min-max search with alpha-beta pruning, derives its heuristics following the strategy described by Winands [35]: concentration, centralization, center-of-mass, quads, mobility, connectedness, and player-to-move.

The network design used with pLoGANN consists of 100 input nodes, a single hidden layer with 20 nodes, and a single node in the output layer for a total of 121 nodes. The activation function g used by every perceptron in the network is the tanh function, a type of sigmoid function:

$$\tanh x = (e^{2x} - 1) / (e^{2x} + 1).$$

A fully connected network, it requires 2000 connections between the input and hidden layers and 20 connections between the hidden layer and output layer for a total of 2020 connections. As each connection has a weight, and each non-input node has a bias weight, there are a total of 2041 weights in the network. Thus, this experiment requires 2041 chromosomes to describe the network.

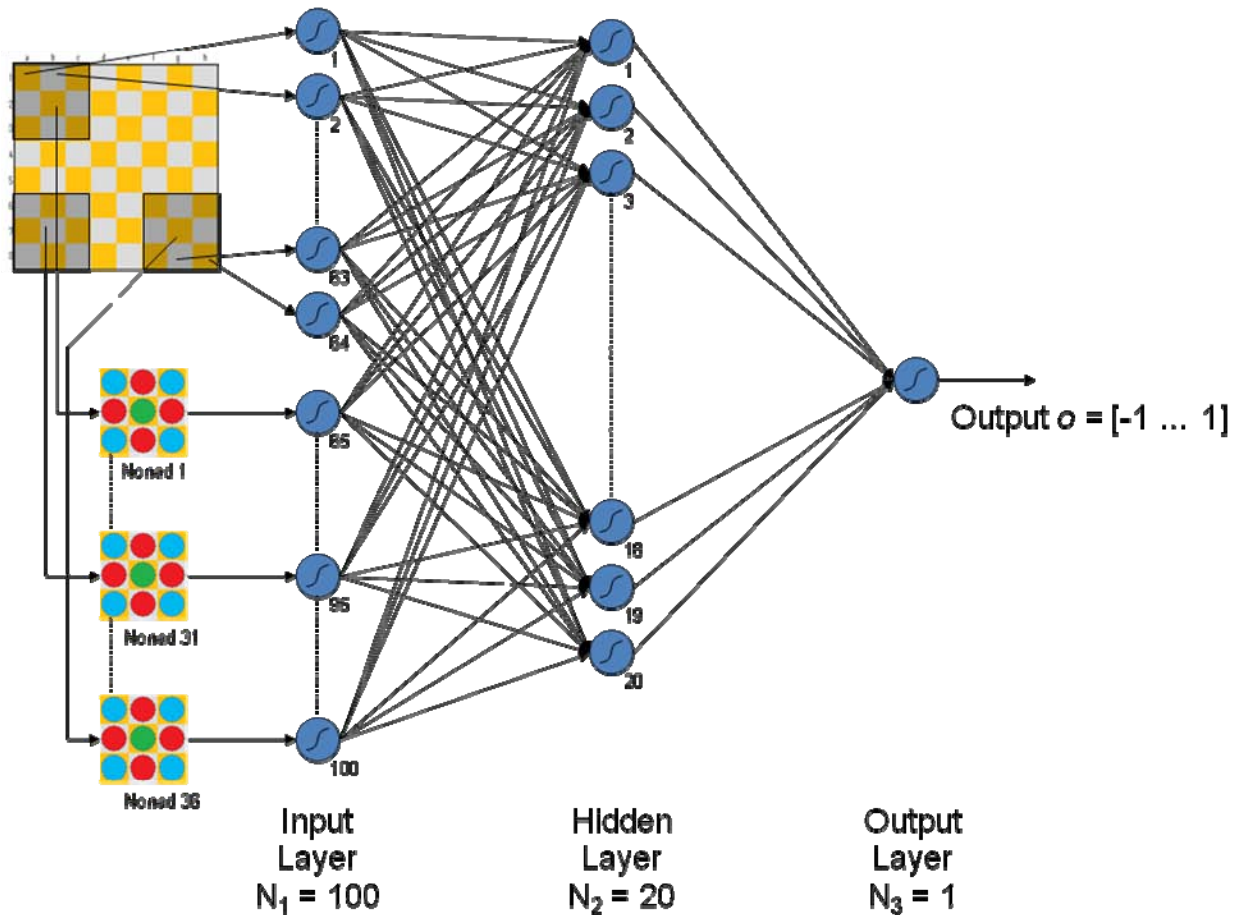


Figure 14: Diagram of pLoGANN's artificial neural network and inputs.

As displayed in the diagram above, the input consists of two sections. The first 64 nodes each receive input from one of the 64 spaces on the board. If the space is occupied by one's own piece, the input value is 0.75; if occupied by one's opponent's piece, the input value is 0.5; otherwise the input value is 0.5. The remaining 36 nodes (nodes 65-100) receive as input the results of the nonad evaluations taken from each of the 36 nonads on an 8x8 board.

This board configuration was chosen for two primary reasons. The first was efficiency. In order to ensure that the algorithm could process enough generations to

provide results within the narrow window of opportunity available for this experiment, a small network was required. Even at this relatively small size, the computer cluster needed to perform the experiment was only available long enough to let the experiment run for approximately ten thousand generations, which is not a lot, considering the size of the solution space. The second consideration was to utilize at least one heuristic that covered the full extent of the board. Rows and columns, diagonals, and self/non-self heuristics on the other hand, require $2n^2 = 128$ inputs. Quadcounts, while requiring only six inputs, do not provide any spatial information. Nonads, on the other hand, provide for full board coverage, including spatial information, and require a much smaller $(n-2)^2 = 36$ inputs. Thus, nonads and the board state itself were chosen as the only heuristics for the final experiment.

Implementation of pLoGANN

As the first initial of pLoGANN's acronym indicates, the final experiment is run in parallel on multiple processors. While development and testing for pLoGANN was performed on a network of Windows PCs running Microsoft Windows XP, the experiment was specifically designed to be conducted on a computer cluster. The cluster used featured thirty-two dual core nodes, each of which is connected via a crossbar, enabling uniform communication. Communication was conducted using network protocols, rather than MPI, which makes pLoGANN technically a distributed program, rather than a parallelized program. The amount of communication relative to the amount of processing is very small. The master only sends out the allele values to each slave, and the slaves return only the identity of the winner of the tournament, which is a single

integer value. As described above, the tournaments and the evaluations versus ALoA are distributed, while the crossover and mutation operations, as well as the persistence of the epochs are performed exclusively by the master.

One of the final choices was to determine to which depth the competitors searched. Search depth, or ply, refers to the number of moves ahead one looks at. For example, if the ply is set to 1, one considers only one's immediate options. If the ply is set to 2, one considers one's immediate options as well as all of one's opponent's options the following turn. The advantage of increasing the ply is that it allows the program to search deeper into the game. However, the disadvantage is that it takes longer to decide upon the move, since one must evaluate many more board states. Given a branching factor of b , a p -ply search requires the evaluation of on average b^p boards. Thus, an increase in ply leads to an exponential increase in search time.

With this in mind, pLoGANN's ply was set to 3. This enabled it to look ahead to avoid any losing moves, if possible, as well as to move towards a potential winning move. A larger ply was not used for two reasons. First, the extra computation cost would have significantly reduced the number of generations computed. Second, the goal was to develop a great heuristic, which would have made a multi-ply search unnecessary, so it was undesirable to allow pLoGANN to have the crutch of additional search depth since that might prevent the evolution of a great heuristic. However, ALoA's search depth was set to 5. This was done because at 3-ply, ALoA is beatable by average quality humans, but is much more formidable at 5-ply.

Stochastic Player

In order to establish a benchmark for performance measurement, an additional experiment is performed. A stochastic player is introduced which makes each move at random, except that board states in which one's own pieces are connected (a winning board state) are rated maximally while board states in which one's opponent's pieces are connected are rated minimally. Thus, the stochastic player still takes advantage of winning moves and avoids losing moves, if possible. The average performance of this player is then compared with the average performance of the chromosomes. Given that each chromosome represents a semi-random location in the solution space, this experiment is intended to measure if the GA has any impact whatsoever in the performance of the ANN. It is possible that the solution space is so jagged that the GA is unable to exploit any surface characteristics. If so, then each chromosome devolves to little more than a deterministic player whose moves are seeded by a random number. Thus, the stochastic player is introduced as a means to determine if there is any measurable improvement in performance over the course of the primary experiment.

Summary

The nature of genetic algorithms is such that there is no clear point at which to terminate an experiment short of finding a true optimal solution. GAs do not exhaustively search a solution space, and so cannot prove that an optimal solution does not exist. Furthermore, in cases where a GA's fitness function is a simulation of what the task that the GA is expected to perform, it may not even be possible to detect when the GA discovers an optimal solution. Thus, the most reasonable termination condition is based

on the real world limitations of the experimenters. In this case, the primary limitation is time, and the secondary limitation is money, in the form of computer resources. For this thesis, both limitations were bottlenecked by the limited access available on the key computer cluster.

In order to perform the major experiment on the parallel cluster, time and processors had to be reserved in advance. Granted approximately a two week window on 32 dual-core nodes, it was imperative to finish developing and testing pLoGANN so that as much of the reserved processor time as possible could be utilized with the actual experiment. As such, preliminary tests on smaller LoA boards and with different neural network configurations were limited and by no means thorough.

Fortunately, most testing and program validation could be performed on regular PCs. Also, smaller numbers of nodes were available for performance testing prior to the commencement of the major experiment. During these tests it was discovered that the master became a bottleneck if it shared a dual-core node with a slave. Consequently, the master resided on its own node and the planned 63 slaves were reduced to 62.

Given the known limit of time and processing power, prior to the experiment, it was known that the upper limit of the number of generations computed was approximately 10,000, for a total of 620,000 competitions, producing up to 62,000 chromosomes which would be evaluated. Given that 62,000 is a miniscule fraction of the number of computable points in the solution space, it was viewed prior to the experiment that the odds of finding a optimal solution was rather slim, if not nil. As such, secondary goals came into focus. The first was to demonstrate improvement via learning. The

second was to determine whether that methodology produced better results than a random player. The former is measured by pLoGANN's results over time while the latter is measured by comparing pLoGANN's and the random heuristic's performances against both shared competition and each other.

IV. Analysis and Results

The results of the tests are used to determine whether or not PLoGANN was a successful in developing a competitive Lines of Action player. As a genetic algorithm, pLoGANN examined many points in the solution space, each one representing separate LoA heuristic. If any of the examined heuristics produced competitive player, then the final measurement was a success. Barring that, a secondary goal was to determine if pLoGANN demonstrated learning. To that end additional tests were designed and run to determine if pLoGANN improved over the course of generations of the genetic algorithm.

Results of pLoGANN

In total, 1000 epochs were computed, each consisting of 10 generations of 248 competitors producing 62 victors. Out of those 620,000 solutions that survived tournament selection, the surviving solutions of the final generation of each epoch were matched against an outside competitor, ALoA, and their performance documented. As each epoch consists of ten generations, 62,000 total solutions competed against ALoA.

In each of the following charts, four values are displayed. The performance of an individual (or set of solutions) while playing black, the performance while playing white, the combined performance, and then the average of the performances while playing black and white. Performance is measured by the results of a match against the traditional LoA heuristic. If the game results in a victory, then it is scored as 100 minus the number of moves required. If it results in a loss, then the score is simply the number of moves the

game lasted. Games that did not end after 50 moves resulted in a score of 50 regardless of the outcome. The match score then is the sum of the scores of each game.

The first objective was to discover a winning player. For that, the best method is to investigate each of the 62 best highest scoring solutions from each epoch:

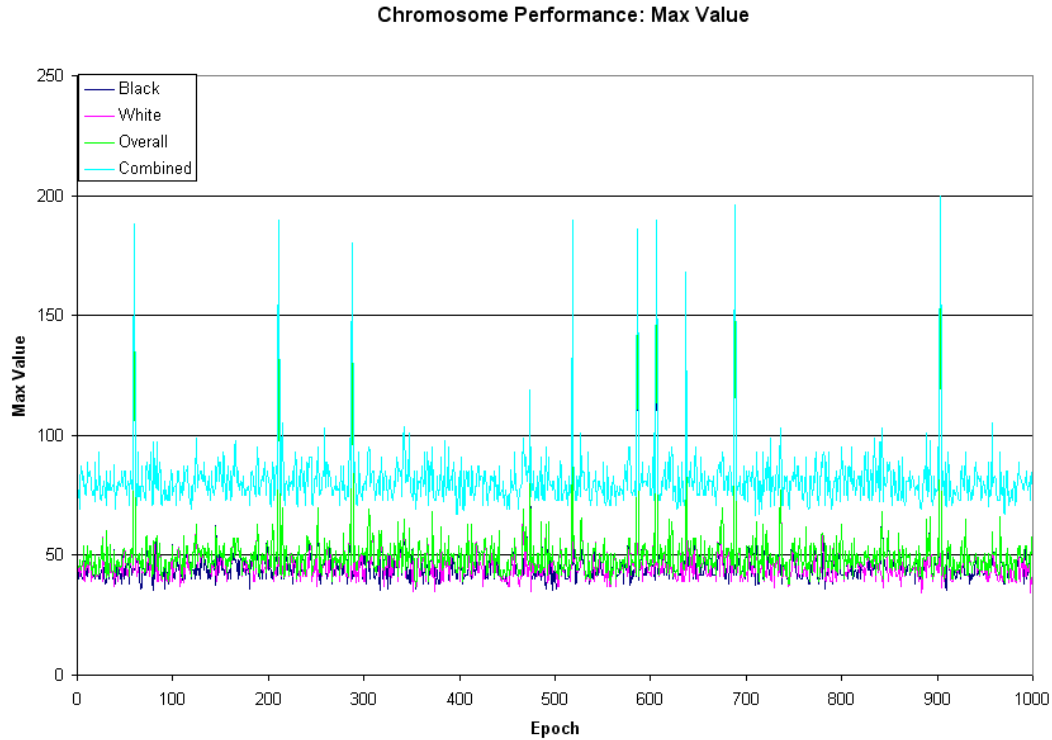


Figure 15: pLoGANN performance of highest scoring solution of each epoch.

As demonstrated in Figure 15, in only nine of the thousand epochs were solutions produced which defeated ALoA (a tenth produced a tie). More significantly, none of those epochs were consecutive, which means the solutions in question did not survive another ten generations of competition, a fact that is examined in further detail in Chapter 5. Furthermore, none of the individual solutions in question were able to defeat a human of average competence. Each of these competitors were victorious playing black or white,

but not both (the highest overall performance came from individual solution 22 of epoch 903 (903:22), which won as black and lost as white in 35 turns each). Thus, it is clear that pLoGANN failed in its primary objective, to produce a competitive Lines of Action player.

The next step is to determine if pLoGANN's failure to find a competitive player is due to limitations of time. In other words, if pLoGANN clearly demonstrated learning, then it stands to reason that given more time (as in more generations), pLoGANN may produce a better player. To measure this, the mean and median values of the epochal solutions are examined. Each represents an 'average' of that epoch's performance.

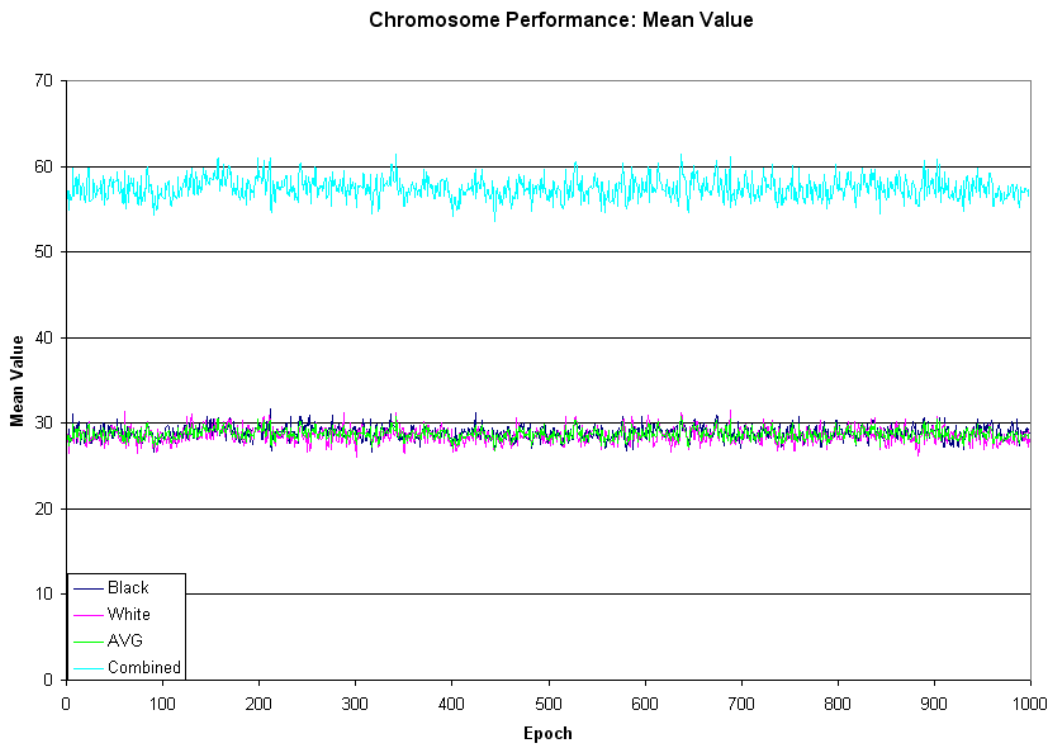


Figure 16: pLoGANN mean performance of each epoch.

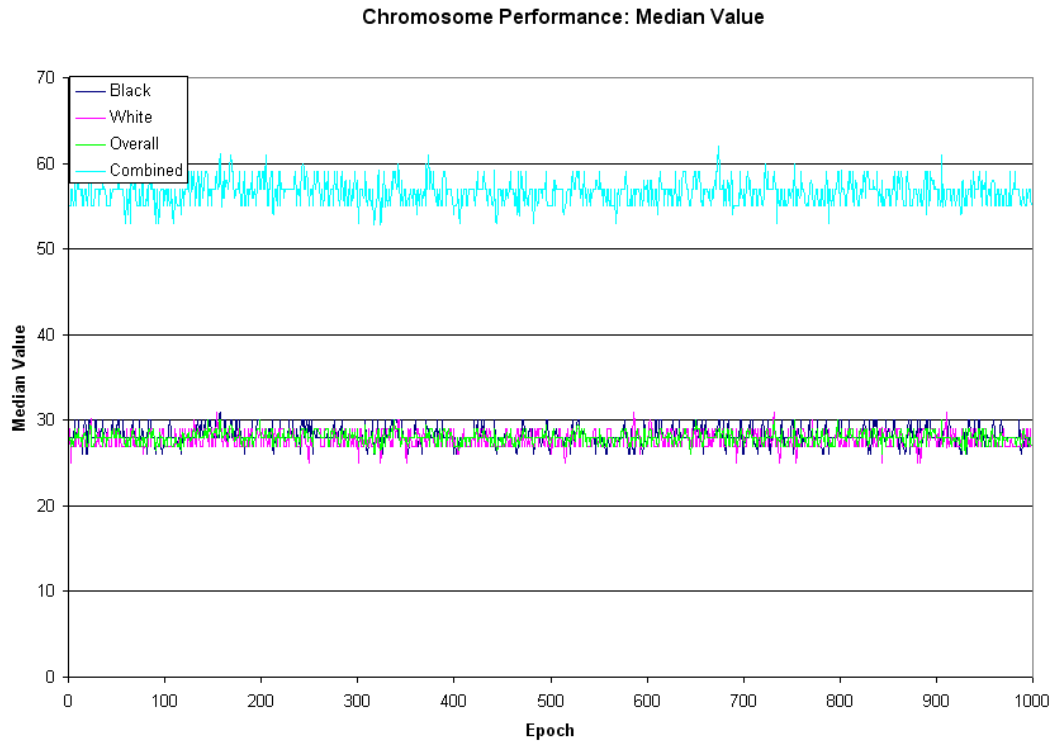


Figure 17: pLoGANN performance of median scoring solution of each epoch.

Both the mean and median graphs show very similar data. In fact, for all four measurements, there is very little difference between the mean and the median in each epoch. In fact, even when an extreme outlier exists, such as in epoch 903, that outlier has only a marginal impact on the mean value, as demonstrated by a comparison of epochs 902-904:

Table 1: Detailed comparison of consecutive epochs

Epoch	Mean	Median	Max	Comb. Mean	Comb. Median	Comb. Max
902	28.26	28	50	56.52	55	83
903	30.43	28	165	60.85	57	200
904	28.47	28	54	56.94	55	85

If solution 903:22 is discounted as a statistical outlier [38], the mean and combined means of epoch 903 would be 29.28 and 58.57, respectively. While this is still higher than the neighboring epochs, the values fall within a single standard deviation (0.60 and 1.20) of the means of all epochs (28.72 and 57.43). In other words, even when an epoch produces what appears to be an outstanding player, the overall performance of that epoch is statistically unexceptional. This trend is more apparent when smoothing is used.

Applying a 5-smoothing average on the results of each epoch with the results of the two preceding epochs and the two following epochs reduces the impact of any outliers such as 903:22. Using 5-smoothing, the data of the chart above looks very similar:

Table 2: Details of consecutive epochs after 5-smoothing is applied

Epoch	Mean	Median	Max	Comb. Mean	Comb. Median	Comb. Max
902	29.05	28.00	89.67	57.77	56.20	108.00
903	29.23	28.00	87.00	57.91	56.20	108.00
904	29.13	28.00	49.33	58.44	57.00	106.00

The 5-smoothing has a similar effect on the graphs produced by all of the epochs, shown below with trend lines for the combined data. As the trend lines demonstrate, there is no indication that the general population of solutions produced by pLoGANN improves during the course of the thousand epochs. In fact, the trend lines have very slight negative slopes, although the negative slope is so small as to be insignificant.

In terms of success at solving the game of Lines of Action, the results of pLoGANN show no evidence of either evolution or devolution. In fact, if anything, the

data suggests that during the first 10,000 generations of its run, pLoGANN behaved like a random player, occasionally coming across an individual that played the right sequence of moves to beat ALoA.

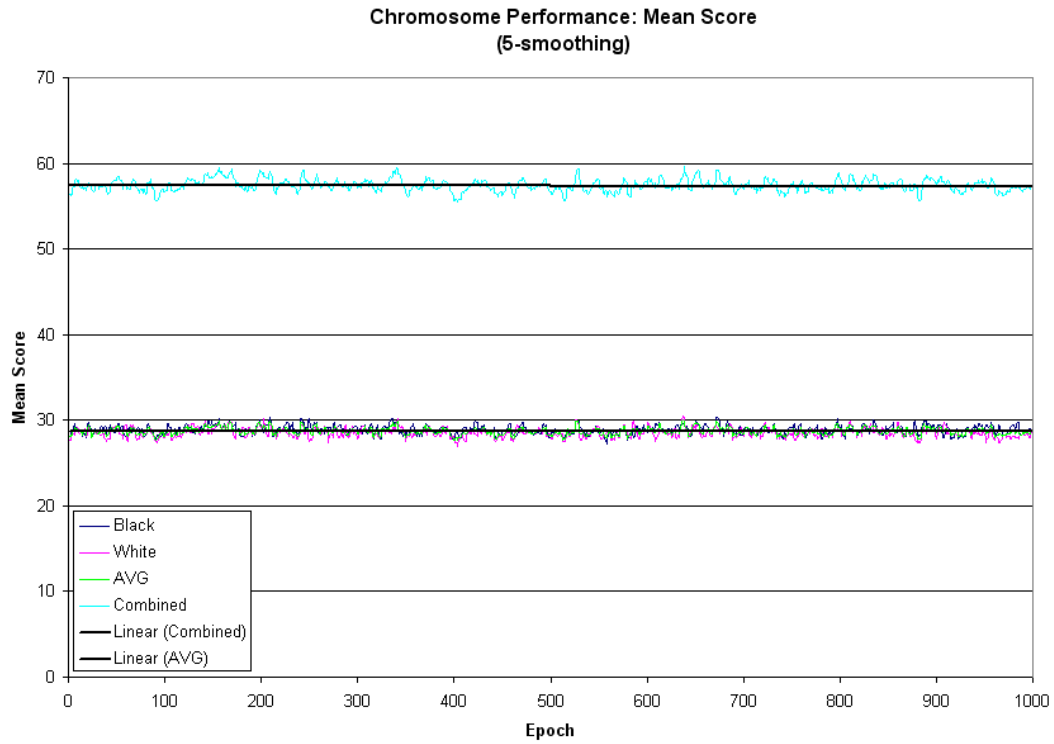


Figure 18: pLoGANN mean performance after 5-smoothing is applied.

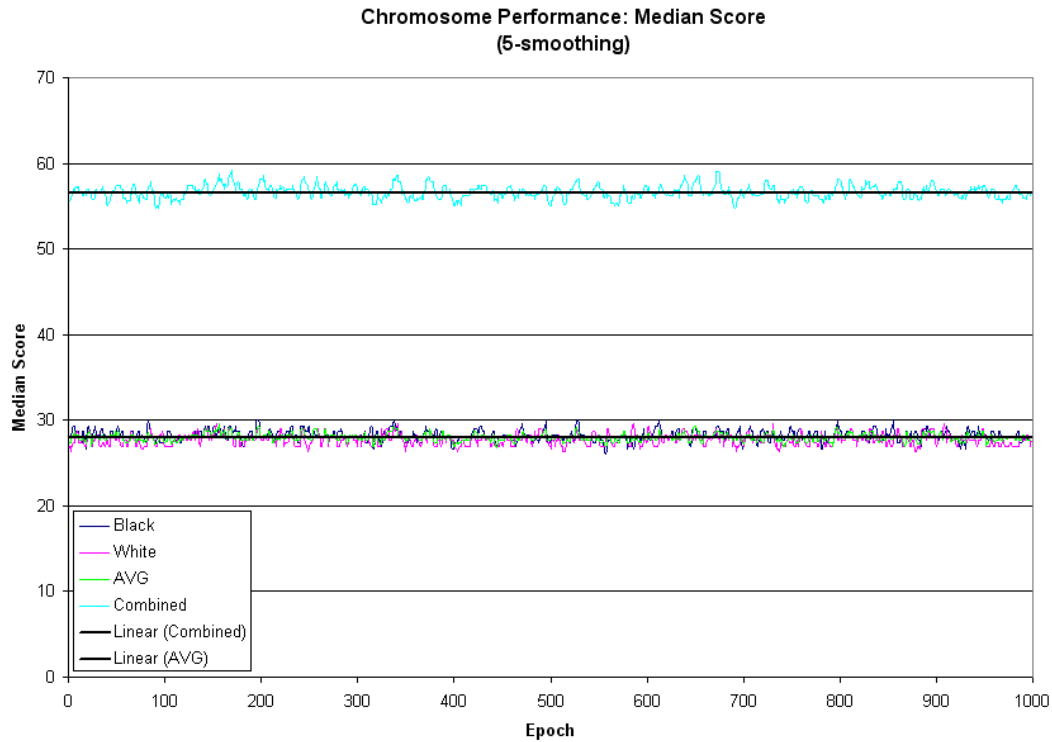


Figure 19: pLoGANN median scoring performance of each epoch after 5-smoothing is applied.

Results of Random Heuristic

To test whether or not pLoGANN did indeed perform like a random player, a stochastic player was developed and tested against ALoA. The stochastic player, or rather the random heuristic, assigns values in one of three ways: a winning move is given the highest rating, a losing move is given the lowest rating, and all other moves receive a rating returned from the pseudo-random number generator from in the default Java Random class.

Table 3: Results of random heuristic

	Overall	White	Black	Combined
Mean	30.98	30.67	31.28	61.95
Median	28	28	29	59
Max	162	100	162	208
Standard Deviation	10.45	10.20	10.39	14.65

The simulation was modeled after pLoGANN’s simulation. The random player played one game as white and then one as black to complete a match, which was repeated 62,000 times. The results are tabulated in Table 3. The combined score pairs the white and black scores from each match. One item of note is that the random heuristic won exactly one game and one match. Thus the max score from black of 162 is an aberration. The next highest black score was 100, a tie game. Also of note is that the random heuristic performed slightly better as black than as white. This is not surprising, given the tendency for the side that moves first in some abstract strategy games, including chess, to have an advantage [39].

Table 4: Comparison of random heuristic and pLoGANN

	<i>Random Heuristic</i>				<i>pLoGANN</i>			
	Overall	White	Black	Combined	Overall	White	Black	Combined
Mean	30.98	30.67	31.28	61.95	28.71	28.55	28.87	57.43
Median	28	28	29	59	28	27	28	57
Max	162	100	162	208	165	164	165	200
St. Dev.	10.45	10.20	10.39	14.65	5.54	5.48	5.48	7.84

By merging all the epochs of the pLoGANN simulation, one can easily compare the performances of the random heuristic and pLoGANN (see Table 4). These results indicate that in some ways pLoGANN behaved very much like the random heuristic. The median scores were nearly identical, while the mean score of the random heuristic

exceeded that of pLoGANN by only 7.9%. Nevertheless, given the very large sample sizes (124,000 for overall, 52,000 for the other three categories), a t-test upon the null hypothesis that the two samples have in fact the same mean returns a zero percent probability for all four categories, firmly rejecting the null hypothesis.

The notable contrast between the two occurs with the standard deviation. pLoGANN's standard deviation is nearly half of that of the random heuristic. The cause is easily discernable when examining the graph of the random heuristic's maximum scores when the matches are divided to epochs in the same manner as pLoGANN (Figure 20).

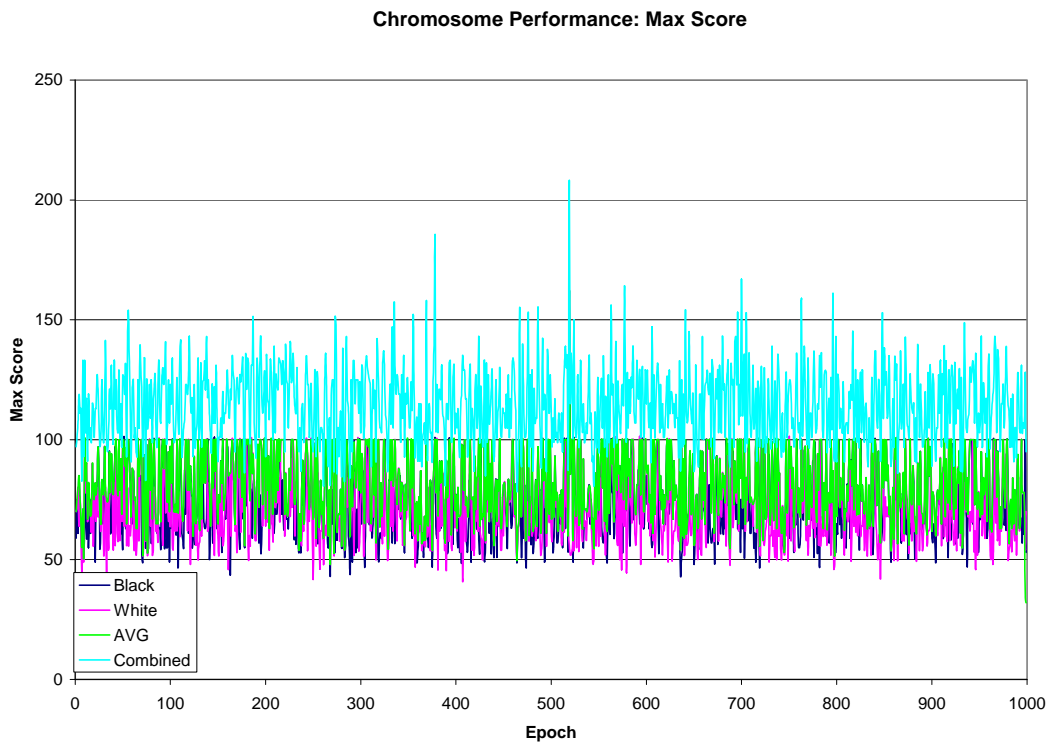


Figure 20: Highest scores achieved by random heuristic within each epoch.

Unlike pLoGANN, the random heuristic frequently forced tie games. These tie games thus pushed up the mean values as well as the standard deviation. Although pLoGANN managed to win seven more matches than the random heuristic, both one and eight victories in 124,000 games should be considered outliers. In the greater scheme of things, the only meaningful difference between the performances of pLoGANN and the random heuristic is the number of tied games the random heuristic achieved.

These results brought up two questions. First, is ALoA so good that there is a *threshold of competence* that must be reached before evidence of learning can be measured? To use a sports analogy, if a novice basketball player plays one-on-one against a professional basketball player for one thousand games, he will in all probability improve as a player during the course of those games. However, the professional player is likely so good that it will take more than just a little improvement before the novice's improved competence is actually measurable based on the game statistics. In that sense, the professional player possesses some threshold of competence that the novice player must surpass before his learning produces improved measurable results in the box score.

Second, if the ALoA possesses such a threshold of competence, does that mean that pLoANN may indeed have learned to play Lines of Action better? If it did not pass that threshold, any improved capability would have been obscured by the dominance of ALoA. For that reason, one more experiment was designed. The most rudimentary heuristic is the random heuristic, which should not have any threshold of competence. Thus, if pLoGANN did indeed learn how to play LoA better, that learning should be evident in competitions against the random heuristic.

Results of Random Heuristic vs. pLoGANN

For this experiment, every tenth epoch was extracted and matched against the random heuristic. Just as each heuristic was set to a search depth of 3 for their matches against the traditional heuristic, so were each set to a search depth of three when matched against one another. The results of these matches are shown below (results show the performance of pLoGANN, not the performance of the random heuristic):

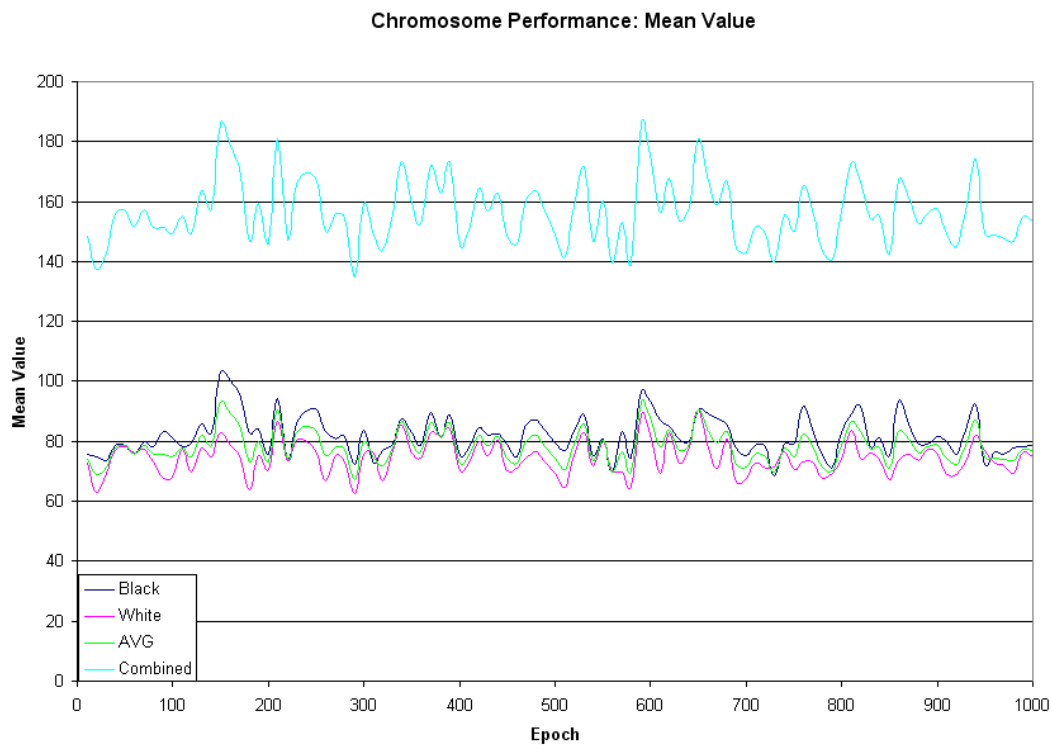


Figure 21: pLoGANN mean performance against random heuristic.

Between epochs 10 and 150, pLoGANN's score for black rises steadily from 75.74 to 103.03. However, after that peak, it descends and no further epoch displays a winning average against the random heuristic while playing either white or black.

Furthermore, the values for the tenth and thousandth epochs all fall well between the minimum and maximum values for any epoch.

Table 5: Results of random heuristic competing against pLoGANN

	Random Heuristic				pLoGANN			
	Overall	White	Black	Combined	Overall	White	Black	Combined
Mean	121.81	125.50	118.13	243.62	78.19	74.50	81.87	156.38
Median	129	133	124.5	252	71	67	75.5	148
Max	186	186	181	353	182	179	182	328
Standard	31.47	30.78	31.36	45.45	31.47	30.78	31.36	45.45

The overall statistics for this experience show that the random heuristic clearly outplayed pLoGANN. Not surprisingly each side's best games were similar, but the random heuristic had significantly higher scores through each epoch compared, with the singular exception of epoch 150 when pLoGANN played as black. Their actual records further demonstrate the dominance:

Table 6: Results of games and matches between random heuristic and pLoGANN

	pLoGANN			
	Wins	Losses	Ties	Winning Percentage
All Games	2168	9089	1143	19.26
White	959	4792	449	16.68
Black	1209	4297	694	21.96
Match	1099	4976	125	18.09

These results, however, only examine networks that have survived at least one round of tournament selection. That leaves the question of the base case scenario: how does a pLoGANN network perform prior to any evolution? To answer that question, a network, dubbed RandomANN, was created, whose weights were set to uniformly distributed random values between -1 and 1. This network then competed in 1240

matches against the random heuristics using the same settings as before (search depth of 3). The results of this experiment are presented in Tables 7 and 8.

Table 7: Results of pLoGANN and RandomANN versus random heuristic

	<i>RandomANN</i>				<i>pLoGANN</i>			
	Overall	White	Black	Combined	Overall	White	Black	Combined
Mean	55.41	52.07	58.75	110.82	78.19	74.50	81.87	156.38
Median	51.5	48	53.5	104.5	71	67	75.5	148
Max	166	159	166	281	182	179	182	328
Standard	21.71	19.63	22.68	32.38	31.47	30.78	31.36	45.45

Table 8: Results of games and matches between random heuristic and RandomANN

	<i>RandomANN</i>			
	Wins	Losses	Ties	Winning Percentage
All Games	38	1180	22	3.12
White	16	599	5	2.60
Black	22	581	17	3.65
Match	17	602	1	2.75

These results provide two insights into pLoGANN. First, the early hypothesis that pLoGANN was essentially a random heuristic does not hold up. In a trial of 62,000 matches between identical random heuristics, the two competitors produced winning percentages of 49.9% and 50.1%. By comparison, RandomANN has a mere 2.75% winning percentage while pLoGANN had an 18% winning percentage against the random heuristic. Thus, there clearly exists some attribute within the network itself, perhaps a dependency between the perceptrons, that prevents it from performing like a random heuristic.

The second insight is that it appears that pLoGANN did in fact improve its performance relative to the RandomANN. However, this improvement actually took

place within the first epoch. So while pLoGANN did learn how to play LoA slightly better, it quickly plateaued and remained a very weak competitor for the remainder of the experiment. As such, not only did pLoGANN fail to learn much, but there is some component of pLoGANN's heuristic that actively leads it to make bad moves. For these moves to place it at such a disadvantage to a random heuristic indicates a significant failure on pLoGANN's part.

Summary

The early experiments which led to the development of pLoGANN produced unclear results. The additional challenges presented by the computational limitations set by the limited access to computing power led to a minimalistic design for pLoGANN that incorporated a speculative heuristic (nonads). However, every experimental result showed that not only did pLoGANN begin the experiment as an extremely weak Lines of Action player, never improved to the point of being able to compete with even a random heuristic. In short, pLoGANN is a failure.

V. Conclusions and Recommendations

The results of the final experiment clearly indicate that pLoGANN failed to develop a quality Lines of Action player. However, the reason it failed to do so is much less clear. In this chapter, several possible reasons why pLoGANN failed are examined. Also, the benefits of the approaches taken towards pLoGANN's design are analyzed, and suggestions for improvements discussed, as well as applications for this approach besides solving Lines of Action.

Conclusions of Research: What Went Wrong

Without a doubt, pLoGANN failed to develop a strong LoA player. The reasons why, however, are ambiguous. There are several factors that may or may not have contributed to the lack of success:

- *Not enough time:* pLoGANN ran for 10,000 generations and in that time revealed no measurable improvement. However, it is always possible that given enough time, an individual solution could be found in a region in the solution space that would enable the population to converge. However, short of a full analysis of the solution space, which is completely infeasible, this is only a supposition, and not worth investigating. pLoGANN, as is, ran for a long time, and its results gave no indication that additional time would produce a better outcome.
- *Existence of a solution:* One unanswerable question is whether or not a quality solution, one which effectively approximates the value of a Lines of Action state exists within the solution space of pLoGANN. This is akin to asking

whether or not the structure of pLoGANN's final tree can be tuned to solve Lines of Action. If not, then pLoGANN was predestined to fail the primary objective. However, even if that were the case, there would still be a best solution within the solution space, and the question would then be how good of a solution, compared to the best achievable solution, did pLoGANN achieve? Given the poor performance of the best measured solutions against human players, if pLoGANN did indeed land at a high quality solution, then clearly this entire approach to solving LoA is invalid.

- *Solution space topography*: Genetic algorithms are essentially hill-climbing mechanisms with the ability to escape local minima. However, in order to climb the hill, there must exist a hill to climb. The topography of pLoGANN's solution space is unclear, and given that it is 2,041 dimensions, mapping the solution space is well beyond the scope of this thesis. If there is in fact no discernible topography, then the task at hand reduces to what in a sense is a needle in the haystack problem. Since pLoGANN examined only a tiny fraction of the solution space, there was little chance of it finding a good solution if indeed there is no useful topography.
- *Crossover scheme*: Even assuming that pLoGANN's solution space has an adequate topography, it is unknown if pLoGANN was capable of following the topography in a useful manner. There is much debate within the GA community [23, 30] about the merits of different crossover schemes. Since each individual solution's chromosomes fit separate weights on the artificial

neural network, and since the ordering of those weights matters when crossover operations occur, the choice of crossover schemes and the ordering of the weights can produce very different results. As a specific example, the input layer was organized linearly, starting at the upper left corner of the board and proceeding like words on a page to the lower right corner (the nonads portion of the input weights were ordered similarly). Thus, weights eight and nine were adjacent, even though the squares they represented (the top-right corner square and the left most square on the second row, respectively) were not connected in any way relative to the game of Lines of Action (i.e., pieces in those two squares cannot directly effect one another during any given turn in Lines of Action, since they do not occupy the same horizontal, vertical, or diagonal lines). On the other hand, either a spiral scheme or a salamander scheme would have ensured that adjacent weights represented adjacent squares. However, given that at some point there would have to be some disconnect between adjacent weights and what they represented, such as the separation between input weights and hidden layer weights, a re-ordering of weights could not have created uniform consistency. pLoGANN had both crossover and mutation schemes in place which would allow it to explore and converge; the fact that it failed to converge at all suggests that different reproductive schemes are unlikely to have produced better results in the face of other problems with pLoGANN.

- *Tournament selection*: It was a known fact at design time that by far the greatest computational bottleneck that pLoGANN faced was the fitness function. As such, there was a deliberate attempt to extract as much utility out of the fitness function as was possible. To that end, it made sense to implement a tournament selection scheme, since that allowed two individuals to be evaluated using one call to the fitness function (the fitness function being a match between the two individuals). Had a different fitness function been used, such as the fitness function to evaluate each epoch (by competing each individual against a traditional heuristic), then two drawbacks would have occurred. The first is that only half as many individuals could have been evaluated. The second is that the de facto objective of the GA would then have been to find a chromosome best able to compete against that heuristic, rather than to find the best general Lines of Action player. This might very well have led to a final population that consisted of solutions similar to the outliers that did indeed defeat the heuristic (especially if the heuristic was deterministic, as was the case with the one used to evaluate the epochs in pLoGANN), yet were clearly unexceptional players as measured by competition against beginner level humans. However, pLoGANN, as implemented, clearly did not succeed, and this remains one of the possible reasons why.
- *Elimination scheme*: The tournament selection model used by pLoGANN used a single elimination format. The advantage of this method is that it

allows a genetic algorithm to examine more individual solutions, since an individual remains in the population if and only if it wins every match. Such an advantage is useful on a tremendously large solution space such as the one pLoGANN operated on, but it comes at the risk of eliminating quality solutions too quickly. For example, a solution that defeats 60% of the population only has a 13% chance of surviving two tournaments. In fact, a solution that defeats 90% of the population has only a 12% chance of surviving an entire epoch. In hindsight, it should come as no surprise then that not a single individual was repeated in any of the 1,000 epochs measured. Since a population is measured against an outside player only once per epoch, it is certainly possible that a great solution was evolved and perished within a single epoch and so was never recorded. However, even if that were so, it is clear by the absence of any rise in the population averages that any such superior solution did not have a lasting impact on the population. During design time, naïve optimism that a perfect solution would be found, and that it would persist across epochs, led to this area being overlooked. Thus, if pLoGANN were to be redesigned, this portion of the program would almost certainly be implemented differently, to allow for greater persistence of solutions.

- *Tournament format:* The simplicity of a single elimination tournament also had the advantage of requiring the fewest possible matches to determine a winner. This efficiency, however, exacerbated the single elimination issues

described above. Another approach, a round robin tournament featuring six total matches amongst four competitors would have rewarded solutions that were better against more opponents. However, this, too, is unclear, especially considering all competitions within pLoGANN were strictly deterministic (unlike real life athletic competitions), and where no game or match was influenced in any way by any events before or after that game or match (again, unlike real-life athletic competitions, where injuries, fatigue, and standings impact in-game decisions). In the single elimination scheme, the winner defeated two of the three opponents. In a round robin scheme featuring the same four participants, however, the tournament would produce a different winner only 25% of the time (see Appendix D), and even then the victor would statistically be only marginally better, as it's victory would be due to a tiebreaker. Thus, the cost of computing only half the generations is pretty steep compared to the gain of what may or may not be a better solution only one quarter of the time.

- *Population size*: During the design of pLoGANN, it was obvious that in order to maximize the efficiency of the parallel environment, that the population size should be some multiple of the number of slaves available, 62. In pLoGANN, that multiple was chosen to be one. The effect this had was that it effectively gave priority to depth of search over breadth of search. Had the multiple been set to any other positive integer, the population would have been much larger, but the number of generations calculated would have been

reduced by a factor of the same multiple. The total number of tournaments held, however, would have remained the same given the same timeframe. The primary concern of having a small population was that it may lead to premature convergence. This, however, clearly did not happen. While a larger population may have produced a different result, the impact of population size is secondary. In other words, a larger population cannot have a significant impact if the basic mechanisms of the GA are not functioning well.

A pair of the above issues can be dismissed outright. There is no indication in any of the evaluations that pLoGANN was learning, and so there is no justification to conclude that more generations would have produced different results. Similarly, there is no evidence that a different population size could have produced better results.

Of the items above, two are issues that arose prior to the development of the genetic algorithm. If no configuration of the ANN could produce a quality LoA heuristic, then pLoGANN was doomed as of the moment the network configuration was determined. Similarly, if the solution space topography was so chaotic that the problem reduced to a “needle in the haystack” problem, then no GA could navigate it successfully. If either of these conditions were true, then no amount of tuning the GA would have produced viable results. However, neither of these problems are feasibly diagnosed, and therefore it is unknown to what extent these factors impacted pLoGANN’s performance.

Two of the six issues that pertain directly to the GA, the choice of using tournament selection and the tournament format no doubt impacted how the GA

operated. However, given the relatively low amount of resources available for the experiment compared to the extremely large size of the solution space, the basis for these two design decisions still hold up despite pLoGANN's performance. As stated above, tournament selection, by allowing the evaluation heuristic to compare two solutions at once, doubled the number of solutions that could be evaluated within the same amount of time. Similarly, the use of a single elimination tournament scheme, as opposed to a round robin scheme, also doubled the number of generations measured, while producing an inferior solution at most one quarter of the time. The four-fold increase in efficiency generated by these two choices outweighs the relatively minor evaluation differences as well as the unknown impact a different selection format might have had.

The final two issues, however, definitely bear greater scrutiny. The crossover scheme, which was single point crossover, was developed without critically analyzing the chromosome encoding scheme. Given three alleles a , b , and c , which are encoded in that order, single point crossover means that alleles a and c can never be retained in a chromosome unless b is also, but b can be paired with one or both of a and c . Thus, if there exist any dependencies between genes, then this single point crossover has uneven impact on the set of alleles because it enables some combinations of genes to be retained but not other combinations.

The elimination scheme, however, is the element of the GA that had the most visible unintended consequences. The single elimination scheme resulted in an epochal turnover rate of one hundred percent. Without a single solution persisting from one epoch to another, pLoGANN had very little chance of converging, since the turnover rate would

have made it prohibitively difficult for a set of solutions to remain in the population for long. Of all the critical issues with the genetic algorithm at the heart of pLoGANN, this is the one that had the clearest impact.

Conclusions of Research: What Went Right

Given the performance of pLoGANN, it is easy to overlook what pLoGANN did well. pLoGANN represented a combination of several different disciplines within computer science. It combined a meta-heuristic search algorithm with artificial neural networks over a parallelized runtime environment. The effectiveness of the search algorithm combined with the ANN is in doubt, but the effectiveness of the parallelization of pLoGANN is not. Furthermore, the fact that it was designed to operate in a platform-independent manner not only enabled for easy testing of the software, but also presents an expandable framework that can be used for further research.

pLoGANN took advantage of a computational model that was embarrassingly parallel. As such, it required little innovation to configure it to run in parallel or in a distributed environment. Nevertheless, several design and post-testing decisions were made to eliminate inefficiencies during the operation. One such decision was to introduce a four-unit tournament selection. Because of the variability in game length, each generation took as long as the longest single tournament. Thus, the effort to normalize the length of the tournaments by introducing additional competitors and transforming it into a multi-competitor tournament ensured any sequence of three matches (aka one complete tournament in pLoGANN) would take at most as long as three generations had pLoGANN's selector consisted of a single match. Given 62 nodes, the probability that a

single tournament consisted of all three of the longest matches of that generations was 5.64×10^{-9} . Assuming probability held up, this decision enabled pLoGANN to save time in every single generation of this experiment.

Technically speaking, pLoGANN was developed as a distributed application rather than a parallel application, as neither the master nor any of the slaves used shared memory. While pLoGANN forsook the opportunity to use shared memory (and thus reduce the volume of communication between the master and the slaves), it gained the ability to be operated on any kind of cluster of computers, including random personal computers connected by a LAN or even the internet. Not only did this enable much easier debugging and testing, given the abundance of open computer labs at AFIT, but it also makes pLoGANN easily modifiable to perform other tasks in a much more flexible environment.

One inefficiency that pLoGANN suffers from is the requirement that generations be synchronized. While this makes evaluating generations and epochs a more straightforward task, it introduced three notable areas of inefficiency. The first was that the master had to wait for all slaves to finish their computations for each generation. The second was that the slaves which finished sooner had to wait for the slaves that finished later. Finally, all the slaves had to wait for the master to perform crossover and mutation operations. If the need to monitor generational (or in pLoGANN's case, epochal) performance is not as strong as it was for pLoGANN, an asynchronous model could be developed, where the master maintains a reserve of solutions that can be sent out to slaves as soon as one is available, then no slave would remain idle. Obviously, as the

number of slaves increases, other performance issues may arise in the master, but this is a framework that could be utilized to run a distributed search on any environment, whether it is LoA, a different abstract strategy game, or other.

Significance of Research

This research effort yielded decidedly underwhelming results. While it did not yield any positive breakthroughs for the study of Lines of Action, it did demonstrate some of the limitations of artificial neural networks. Lines of Action is not among the most complex of abstract strategy games, but some of its mechanisms are decidedly different than other well known games. The ambiguous value of having more or less pieces on the board is virtually unique, and the absence of direction on the game board adds another challenge, since no side can be weighted higher or lower than any other side (unlike, checkers, for example). Despite this, compared to almost any real-time strategic environment, such as a battlefield or soccer field, Lines of Actions is a very simple environment, and one which an artificial neural network was unable to model in a simple manner. Even with just one hundred inputs and a single hidden layer, the ANN had 2,041 individual weights. When a metaheuristic, such as a genetic algorithm, is used to optimize these weights, it produces a search space so large that a complete search is an intractable problem. If a GA is unable to feasibly train an ANN in a static environment, it does not have any chance of doing so in a dynamic environment, which any real time environment is. This thesis, if nothing else, exhibits some of the limitations of GA-tuned artificial neural networks.

Final Remarks

The continuing advancements in computer engineering are increasing the computational power available to researchers every year. With these resources at their disposal, artificial intelligence continues to be a focus of sophisticated approaches to design machines that can outperform humans. At present, such efforts have produced the best competitors in the world in games such as checkers and Lines of Action. LoA is a niche game with a relatively small following. Research into chess and Go, however, have the potential to capture the general public's attention as well as impact how those games are played at the professional levels as computers introduce new strategies and provide proof or disproof of existing strategies' viabilities. Once artificial algorithms begin to demonstrate the ability to find inefficiencies in existing methods of addressing competitive tasks, whether in games, finance, or military matters, their value can only increase.

Appendix A

Derivation of Parallelization of Fitness Function

According to Gustafson's law, the speedup of parallelizing an algorithm can be given as $S(P) = P - \alpha * (P - 1)$, where P is the number of processors in the parallel implementation, α is the non-parallelizable, or serial, portion of the algorithm, and $S(P)$ is the speedup achieved over P processors [26]. Speedup itself is defined as the improvement in performance of a parallel algorithm over the same algorithm performed sequentially, and is given as $S(P) = T_l/T_p$, where T_l is the performance of the serial algorithm and T_p is the performance of the same algorithm once parallelized. Given that T_l can be expressed using big-O notation as $O(xn)$, where x is the performance of a single instance of a fitness function and n the number of fitness functions that must be performed, the two speedup formulas can be combined as follows:

$$S(P) = T_l/T_p; S(P) = P - \alpha * (P - 1)$$

$$T_l/T_p = P - \alpha * (P - 1)$$

$$O(xn)/O(T_p) = P - \alpha * (P - 1)$$

$$O(T_p) = O(xn)/O(P - \alpha * (P - 1))$$

Given that α is not only a constant, but is in fact 0, since there is no non-parallel portion of any instance of the fitness function, this formula can be reduced to:

$$O(T_p) = O(xn)/O(P)$$

$$O(T_p) = O(xn/P)$$

However, Gustafson's law specifically ignores the cost of communications, which is typically given as t_c and is applied for each instance of communications to a processor, in this case P :

$$O(T_P) = O(xn/P) + Pt_c$$

$$O(T_P) = O(xn/P + Pt_c)$$

Thus, the performance of the a fitness function over a parallel system is given as $O(xn/P + Pt_c)$.

Appendix B

Theorem: Not all valid Board States of Lines of Action are Reachable

Proof by contradiction: Consider the following legal Lines of Action board state:

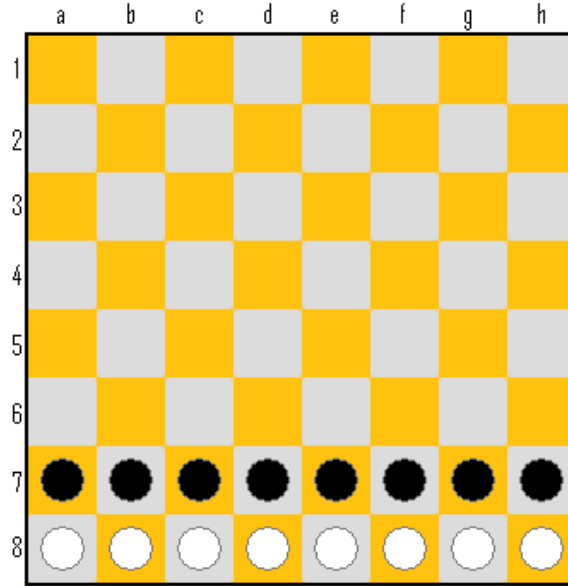


Figure 22: Example of unreachable board state.

If it was black's turn, then white must have made the previous move. However, there exist no moves which white could have made that would have allowed a piece north of the black formation to move south of the formation. Therefore, only black could have made the last move. At the start of black's turn, only two possibilities exist: either it landed on a white piece or it did not. If it did not land on a white piece, then the game must have already been over, since white was already fully connected. If it did land on a white piece, that white piece must have been on one of the eight tiles presently occupied by a black piece, all of which connect to the white formation. Therefore, prior to black's

move, white must have already been fully connected and the game thus already concluded.

As there exists no legal move within the context of a LoA game that reaches the above legal board state, it is proof that not every legal board state is reachable.

Appendix C

Discussion of Offline and Online Reinforcement Learning

The decision to use online reinforcement learning with pLoGANN was based on the advantages and disadvantages compared to offline reinforcement learning. This appendix introduces describes these approaches.

In order to identify a quality network structure offline reinforcement learning is used. It has the primary advantage of speed, which enables the testing of multiple network structures in a relatively short amount of time. Consequently, different structures are designed, tested, and compared to one another with the purpose of incorporating elements of the more successful structures into a final network structure to be used in the final online experiment.

Offline learning is accomplished by sampling the state space and assigning values to each state sampled. This is done by repeatedly playing random games and evaluating board states based on the outcome of each game. For example, in a particular game, if a state leads to a black win, then it is evaluated positively for black, and negatively for white. Each board state receives a cumulative evaluation, based on temporal difference learning [37]. The update of the value for each board state is

$$\lambda_t = V_t(s) - \gamma V_{t+1}(s)$$

where λ_t is represents the difference between the actual value of a state ($V_t(s)$) and the expected value ($\gamma V_{t+1}(s)$), which is the product of the actual value of the next state in the sequence and the discount factor γ , where $0 \leq \gamma < 1$. As the value of γ approaches 1, the difference between the actual and expected values tends to increase.

In the offline learning algorithm, each game is evaluated one at a time. The final position is evaluated as 1 for a win and 0 for a loss. Iterating in reverse order over all the positions of the game, the temporal difference formula is applied to each board state. Thus, each board state, which begins with a neutral evaluation, has its rating appreciated or depreciated slightly depending on whether that board state resulted in a win or a loss for the side in question.

The goal of the algorithm is to slowly have each state reach an equilibrium value. In other words, as more games are played, certain states, particularly early game states, appear over and over again. If each appearance precedes the same outcome, then the value of that state slowly equalizes towards the value of that outcome, either a 0 (loss), 0.5 (tie), or 1 (win). In theory, as more games are played, the valuations of all board states eventually equalize to either a 0, 0.5, or 1, because that state should always lead to either a loss, tie, or win (in games that do not allow for a tied result, all states should equalize to a 0 or 1).

Unfortunately, this is only possible in very small games where it is feasible to visit and store every state. As the state space of LoA is well beyond the capacities of current processing and memory technologies, the practice of sampling is used. By sampling actual games, only samples of achieved board states are taken. Furthermore, sampling of the board states only happens once, before any ANNs are even introduced into the process. The same samples are then stored in a hash table and used for each ANN, which is where this process gains its speed, since half of each subsequent experiment has already been performed by the first experiment.

Once the state space has been sampled, these values may be used to train a neural network (or other construct) using feed-forward/back-propagation. Since the same sampled set is maintained, many different neural networks may be trained and evaluated. Thus, this allows for fast comparison of different network structures. Furthermore, tests are easy to repeat since the network can simply be re-randomized using a different random seed. Finally, the resulting networks are used to play opponents to determine if offline training actually results in a successful player.

Whereas offline learning plays many games and then trains as many networks as desired on the same training set, online reinforcement learning trains the network as games are played. The key benefit of this approach is that it uses much less memory. Offline training requires a databank of board states and their estimated values. Online training only requires that the current state of the network be stored. Given the size of the state space for Lines of Action, this is not a trivial difference. However, the savings in memory come at the cost of operating time. In pLoGANN, the networks perform three ply searches, meaning for each move, they evaluate hundreds of board states. Furthermore, no data may be migrated from experiment to experiment, which means that each repeat must begin at the first step, leading to a higher operating time per experiment when multiple runs are made on the same network, as compared to offline reinforcement learning.

Appendix D

Analysis of Single Elimination versus Round Robin Tournament

The question of whether or not a round robin tournament might produce a different outcome compared to a single elimination tournament is an active subject of debate. In the United States, NCAA athletics adopt single elimination tournaments for many of their championship events, as do all of the major professional team sports (MLB, NBA, NFL, and NHL), while many international soccer tournaments incorporate a hybrid system. The UEFA and World Cup finals, for example, feature a set of four team round robin tournaments in the group stage, the top two finishers of which are then seeded into a single elimination final stage tournament. The merits of the styles are a matter of subjective taste as anything else. However, there is also a component driven by limitations. Because MLB, NBA, NHL postseason matches can last well over a week (each typically consists of a best of seven series that includes days off), adopting a round robin system would significantly alter scheduling.

When it comes to pLoGANN, a couple points stand out. The first is that when four participants are included, a single elimination tournament requires three matches, while a round robin tournament requires six matches. Since the tournaments are the major bottleneck of pLoGANN, in order to justify a switch to a round robin format, there must be ample evidence that such a switch would produce a different outcome, that the different winner would be more deserving, and that having the new winner would justify the costs.

The winner of the single elimination tournament defeated two opponents. In a round robin system, two victories is a substantial leg up towards winning a tournament. Suppose a single elimination tournament consisted of participants a , b , c , and d , where the winner of each match is in red:

a	
b	a
c	d
d	

If those teams were in a round robin tournament, and the matches are deterministic, as is the case in pLoGANN, then the tournament would look as follows:

a	a	a
b	d	c
c	c	b
d	b	d

If one assumes that all participants are about equal and thus have a 50% chance of defeating any given opponent, then a defeats c with probability 0.5 and wins the round outright. However, if c wins, then the outcome is determined by a tiebreaker. The typical first tiebreaker is head-to-head. Thus, if c also defeated b in round two, c would defeat a because of the head-to-head victory. If b defeated c in round 2, then the winner of the b - d match in round three would finish 2-1 and then lose the tiebreaker to a . Thus, a only loses the round robin if c defeats both a and b , which occurs with probability $0.5^2 = 0.25$. Thus,

the winner of the single elimination tournament of deterministic matches also wins the round robin equivalent three times out of four.

Bibliography

- [1] Z. Michalewicz, D. B. Fogel. *How to Solve it: Modern Heuristics, Second Edition*. Heidelberg, Germany: Springer-Verlag, 2004.
- [2] J. Tromp, G. Farnebäck. "Combinatorics of Go." In Proceedings of 5th International Conference on Computer and Games. Torino, Italy, 2006.
- [3] S. Sackson. *A Gamut of Games*. New York, NY: Random House, 1969.
- [4] L. V. Allis. "Searching for Solutions in Games and Artificial Intelligence." Ph. D. Thesis. Rijksuniversiteit Limburg, Maastricht, The Netherlands, 1994.
- [5] T. M. Mitchell. *Machine Learning*. Boston, MA: WCB McGraw-Hill, 1997.
- [6] G. Tesauro. "Temporal Difference Learning and TD-Gammon." *Communications of the ACM* 38, 1995.
- [7] D. B. Fogel. *Blondie 24: Playing at the Edge of AI*. San Francisco, CA: Morgan Kaufmann, 2001.
- [8] J. Schaeffer, N. Burch, Y. Björnsson, A. Kishimoto, M. Müller, R. Lake, P. Lu, S. Sutphen. "Checkers Is Solved." *Science*, 2007.
- [9] "Folding@Home Distributed Computing." <http://folding.stanford.edu/>
- [10] J. M. Thompson. "Defining the Abstract." Unpublished article. <http://www.thegamesjournal.com/articles/DefiningtheAbstract.shtml>, 2000.
- [11] S. Reisch. "Gomoku is PSPACE-complete." *Acta Informatica* 13. 1980
- [12] M. P. D. Schaff, M. H. M. Winands, J. W. H. M. Uiterwijk, H.J. van den Herik, M. H. J. Bergsma. "Best Play in Fanorona leads to Draw." *New Mathematics and Natural Computation* 4, 2008.
- [13] M. H. M. Winands. "Informed Search in Complex Games." Ph. D. Thesis, Maastricht University, Maastricht, The Netherlands, 2004.
- [14] S. Yen, J. Chen, T. Yang, S. Hsu. "Computer Chinese Chess." *International Computer Games Association Journal* 27, 2004.
- [15] C. Cox. "Analysis and Implementation of the Game Arimaa." Masters Thesis, Maastricht University, Maastricht, The Netherlands, 2006.
- [16] J. von Neumann, O. Morgenstern. *Theory of Games and Economic Behavior*. Princeton University Press, 1944.
- [17] M. H. M. Winands, "Analysis and Implementation of Lines of Action." Masters Thesis, Maastricht University, Maastricht, The Netherlands, 2000.
- [18] L. V. Allis, H. J. van den Herik, I. S. Herschberg. "Which Games Will Survive?" *Heuristic Programming in Artificial Intelligence 2: the second computer Olympiad*. Ellis Horwood, Chichester, 1991.
- [19] W. S. McCulloch, W. Pitts. "A Logical Calculus of the Ideas Imminent in Nervous Activity." *Bulletin of Mathematical Biophysics*, Vol 5, 1943.
- [20] F. Rosenblatt. "The Perceptron: A Probabilistic Model for Information Storage and Organization in the Brain." *Psychological Review*, Vol 65, 1958.
- [21] S. Russell, P. Norvig. *Artificial Intelligence: A Modern Approach, Second Edition*. Upper Saddle River, NJ: Pearson Education, Inc., 2003.

- [22] J. H. Holland. *Adaptation in Natural and Artificial Systems*. Ann Arbor, MI: University of Michigan Press, 1975.
- [23] G. Syswerda. "Uniform crossover in genetic algorithms." *Proceedings of the Third International Conference on Genetic Algorithms*, 1989.
- [24] Y. Bengio, Y. Le Cun. "Scaling Learning Algorithms Towards AI." 2007.
- [25] T. Bäck, D. B. Fogel, T. Michalewicz. *Evolutionary Acomputation 1: Basic Algorithms and Operators*. Bristol, United Kingdom: Institute of Physics Publishing, 2000.
- [26] J. L. Gustafson. "Reevaluating Amdah,'s Law." *Communications of the ACM* 31, 1988.
- [27] M. Enzenberger. "Evaluation in Go by a Neural Network Using Soft Segmentation." *10th Advances in Computer Games conference*, 2003.
- [28] T. Bäck, D. B. Fogel, T. Michalewicz. *Evolutionary Acomputation 2: Basic Algorithms and Operators*. Bristol, United Kingdom: Institute of Physics Publishing, 2000.
- [29] C. R. Reeves, J. E. Rowe. *Genetic Algorithms – Principles and Perspectives: A Guide to GA Theory*. Boston, MA: Kluwer Academic Publishers, 2002.
- [30] K. M. Burjorjee. "A Unified Explanation for the Adaptive Capacity of Simple Recombinative Genetic Algorithms." Ph. D. Thesis, Brandeis University, Waltham, MA, 2009.
- [31] D. E. Goldberg, K. Deb. "A Comparative Analysis of Selection Schemes Used in Genetic Algorithms." 1991.
- [32] D. Billings, Y. Björnsson. "Search and Knowledge in Lines of Action." *Advances in Computer Games: Many Games, Many Challenges*, 2004.
- [33] E. Alba, J. F. Aldana, J. M. Troya. *Full Automatic ANN Design: A Genetic Approach, New Trends and Neural Computation*. 1993.
- [34] K. O. Stanley, R. Miikkulainen. "Efficient Reinforcement Learning through Evolving Neural Network Topologies." *Proceedings of the Genetic and Evolutionary Computation Conference*, 2002.
- [35] M. H. M. Winands, H. J. van den Herik, J. W. H. M. Uiterwijk. "An Evaluation Function for Lines of Action." 2004.
- [36] M. H. M. Winands, H. J. van den Herik, J. W. H. M. Uiterwijk. "The quad heuristic in Lines of Action". *ICGA Journal*, 24(1), 2001
- [37] R. Sutton. "Learning to Predict by Methods of Temporal Differences." *Machine Learning* 3, 1988.
- [38] J. S. Milton, J. C. Arnold. *Introduction to Probability and Statistics, Fourth Edition*. Boston, MA: McGraw-Hill, 2003.
- [39] W. F. Streeter. "Is the First Move an Advantage?" *Chess Review*, 1946.
- [40] C. Blum, K. Socha. "Training Feed-Forward Neural Networks with Ant Colony Optimization: An Application to Pattern Classification." *IRIDIA – Technical Report Series*, 2005.
- [41] I. Ghory. "Reinforcement Learning in Board Games." 2004.

REPORT DOCUMENTATION PAGE				Form Approved OMB No. 074-0188	
<p>The public reporting burden for this collection of information is estimated to average 1 hour per response, including the time for reviewing instructions, searching existing data sources, gathering and maintaining the data needed, and completing and reviewing the collection of information. Send comments regarding this burden estimate or any other aspect of the collection of information, including suggestions for reducing this burden to Department of Defense, Washington Headquarters Services, Directorate for Information Operations and Reports (0704-0188), 1215 Jefferson Davis Highway, Suite 1204, Arlington, VA 22202-4302. Respondents should be aware that notwithstanding any other provision of law, no person shall be subject to a penalty for failing to comply with a collection of information if it does not display a currently valid OMB control number.</p> <p>PLEASE DO NOT RETURN YOUR FORM TO THE ABOVE ADDRESS.</p>					
1. REPORT DATE (DD-MM-YYYY) 25-03-2010		2. REPORT TYPE Master's Thesis		3. DATES COVERED (From – To) June 2005 – March 2010	
4. TITLE AND SUBTITLE EVOLUTIONARY ARTIFICIAL NEURAL NETWORK WEIGHT TUNING TO OPTIMIZE DECISION MAKING FOR AN ABSTRACT GAME				5a. CONTRACT NUMBER	
				5b. GRANT NUMBER	
				5c. PROGRAM ELEMENT NUMBER	
6. AUTHOR(S) Miller, Corey M.				5d. PROJECT NUMBER ENG09-219	
				5e. TASK NUMBER	
				5f. WORK UNIT NUMBER	
7. PERFORMING ORGANIZATION NAMES(S) AND ADDRESS(S) Air Force Institute of Technology Graduate School of Engineering and Management (AFIT/EN) 2950 Hobson Way, Building 640 WPAFB OH 45433-8865				8. PERFORMING ORGANIZATION REPORT NUMBER AFIT/GCS/ENG/10-06	
9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES) Air Force Research Labs, Sensor Directorate, Reference Branch Attn: Dr. Jacob Campbell (jacob.campbell@wpafb.af.mil) 2241 Avionics Circle, Area B, Bldg 620, Wright-Patterson AFB, OH (937)785-6127x4154				10. SPONSOR/MONITOR'S ACRONYM(S) AFRL/RYRN	
				11. SPONSOR/MONITOR'S REPORT NUMBER(S)	
12. DISTRIBUTION/AVAILABILITY STATEMENT APPROVED FOR PUBLIC RELEASE; DISTRIBUTION UNLIMITED.					
13. SUPPLEMENTARY NOTES					
14. ABSTRACT Abstract strategy games present a deterministic perfect information environment with which to test the strategic capabilities of artificial intelligence systems. With no unknowns or random elements, only the competitors' performances impact the results. This thesis takes one such game, Lines of Action, and attempts to develop a competitive heuristic. Due to the complexity of Lines of Action, artificial neural networks are utilized to model the relative values of board states. An application, pLoGANN (Parallel Lines of Action with Genetic Algorithm and Neural Networks), is developed to train the weights of this neural network by implementing a genetic algorithm over a distributed environment. While pLoGANN proved to be designed efficiently, it failed to produce a competitive Lines of Action player, shedding light on the difficulty of developing a neural network to model such a large and complex solution space.					
15. SUBJECT TERMS					
16. SECURITY CLASSIFICATION OF:			17. LIMITATION OF ABSTRACT	18. NUMBER OF PAGES	19a. NAME OF RESPONSIBLE PERSON Gilbert L. Peterson, PhD
a. REPORT	b. ABSTRACT	c. THIS PAGE			19b. TELEPHONE NUMBER (Include area code) (937) 255-6565, ext 4281 (gilbert.peterson@afit.edu)
U	U	U	UU	98	